

Your Favorite Programs from RUN

RE RUN

VOLUME TWO

Disk \$19.97



C-64	page	VIC-20	page
"SPRITECD"*	1	"20FILTER"***	4
"SPRITECD/DEMO"		"20SQUEEZER"***	7
"64FILTER"	4	"20EXPANDER"	11
"64SQUEEZER"	7	"MULTICOLOR"***	56
"64EXPANDER"	11	"SCREEN RELOCATOR"***	
"DATAFILE"	14	"PONIES"	63
"DFMAIL"	24	"SPACE"	66
"DFREPORT"	29	"SPACEMOD"	
"BATTLESHIP"	36	"PRESIDENT"***	69
"SLIDE"	41	"20TOUCHDOWN"***	50
"SLIDEMOD"		"NIMBOTS"***	71
"MANOR"	45	"FRIEND"***	76
"MONEY"	47		
"MONEYMOD"			
"64TOUCHDOWN"	50		
"SPELLER"	53		

*Must be loaded with LOAD"SPRITECD",8,1

**Require a 3K expander

***Require an 8K expander

(remember the asterisks are not part of the program name)

ReRun UPDATES:

Please note:

Included on the ReRun disk/tape are two programs not mentioned in the directory. SPRITECD/DEMO, which is discussed in the article 'Sprite Control' on page 1 and SCREEN RELOCATOR discussed in the article 'The Many-Colored VIC' on page 56.

The article 'Sprite Control' talks about a basic loader which is used to create SPRITECD; we have already created and saved SPRITECD for you on the disk/tape.

After a little checking we discovered that some of the memory requirements listed in both the articles and directory were misleading. On the back side of this sheet is an updated directory stating the correct memory configurations needed. (If you have questions check the articles.) It should also be noted that you should make copies of programs such as SQUEEZER, EXPANDER, DATAFILE, DFMAIL, DFREPORT, and MULTICOLOR onto a separate diskette or tape as these programs use SAVE and LOAD commands that may eventually fill up (or even destroy) your ReRun diskette or tape.

One final note. The programs 64SQUEEZER, 20SQUEEZER, 64EXPANDER, 20EXPANDER, DATAFILE, DFMAIL, and DFREPORT

will only work on a disk system.

The program MULTICOLOR was written for tape system. See the article for saving and loading procedures.

We have included these on the tape for your future use. We are continually checking and rechecking the programs on ReRun to insure you get the very best that RUN has to offer. (In some cases the programs are better than those in the magazine because of improvements and corrections sent in after publication) If you have any problems or suggestions we invite your input.

Guy Wright



DIRECTORY

C-64	PAGE	VIC-20	PAGE
"SPRITECD"*	1	"20FILTER"***	4
"64FILTER"	4	"20SQUEEZER"***	7
"64SQUEEZER"	7	"20EXPANDER"***	11
"64EXPANDER"	11	"MULTICOLOR"	56
"DATAFILE"	14	"PONIES"	63
"DFMAIL"	24	"SPACE"	66
"DFREPORT"	29	"SPACEMOD"	
"BATTLESHIP"	36	"PRESIDENT"***	69
"SLIDE"	41	"20TOUCHDOWN"	50
"SLIDEMOD"		"NIMBOTS"***	71
"MANOR"	45	"FRIEND" **	76
"MONEY"	47		
"MONEYMOD"			
"64TOUCHDOWN"	50		
"SPELLER"	53		

* These are machine language programs that must be loaded with LOAD"name",8,1 for disk or LOAD"name",1,1 for tape. Please remember, the asterisk is not part of the file name.

** These programs require 3K expander.

*** These programs require 8K expander .

C-64 and VIC-20 are registered trademarks of Commodore Business Machines, Inc.

Design/ Production

Laurie MacMillan, Carol Woodman



Introduction

The long-awaited 2nd edition of ReRUN is finally here. The response to the first edition was so positive that we are working on a number of special RUN projects beginning in 1985. To wrap up our first year we are putting as many of the best programs we could possibly squeeze on the 2nd edition ReRUN tapes and diskettes. Games to amuse you, utility programs to help you get the most out of your Commodore computer, and educational programs for young and old. As you know, there were a lot of good programs in RUN magazine from July to December, and we tried to take the best of the lot for ReRUN, but we just couldn't fit them all in. Weeks went into the selection process with various staff members fighting for their personal favorites. We think that the programs selected are among the very best, and I hope you agree.

For you gamers, there are joystick gems such as Money Grubber (C-64), and Lost In Space (VIC). Quick-keyboard

contests like Battleship War (C-64), a fast-paced battle on the spritely high seas. And for anyone who ever wanted to call the plays but avoid the bruises, there is Touchdown (C-64 and VIC). For the thinking gamer there are a number of amusements. Nimbots (VIC) is an ancient take-away game with some new twists. Slide (C-64) may be one of the toughest challenges you ever slip into. Head for the track in Playing the Ponies, and enjoy horse racing action without losing your shirt. Then, somewhere between joystick action and deductive reasoning there is Mystery of Lane Manor, a multi-player solitary detective game for every amateur sleuth in your house. And just for fun we included I Am the President (VIC), a humorous conversation with an infamous ex-president.

For the utility-minded Commodore users, we have programs that will make your programming life easier. Sprite Control (C-64) gives you four new Basic commands for manipulating sprites. The

Many-Colored VIC helps take the frustration out of multi-color character programming. Lister Filter (C-64 and VIC) will let you print out those listings, automatically translating all those graphics and control characters into a form that humans can understand.

Line Squeezer / Line Expander (C-64 and VIC) are two programs for the price of one, to squeeze those long programs down to size or take those scrunched programs and spread them out. Finally, Datafile (C-64), a full-blown database, offers features found in professional software.

No matter what you do with your C-64 or VIC-20, there is something on ReRUN to use, amuse, fascinate, educate, enlighten, solve, battle, help, provoke, hear, see, play, and value. No typing, no trouble. Load them into your Commodore computer and enjoy the best from RUN magazine July-December 1984.

*Guy Wright
Technical Editor
RUN magazine*



HOW TO LOAD

DISK:

To load any of the programs type:

LOAD "program-name",8

then press the RETURN key.

The disk drive should 'whirr' while the screen prints SEARCHING FOR program-name. The screen should then print LOADING and then finally READY with the flashing cursor beneath. Type RUN and press the RETURN key. The program will then begin.

CASSETTE:

Insert the cassette tape into the Datasette recorder with the proper side facing up (Commodore 64 side up if you own a Commodore 64 and VIC-20 side up if you own a VIC-20)

Make sure that the tape is rewound all the way to the beginning.

Type

LOAD "program-name"

then press the RETURN key. The screen will display

PRESS PLAY ON TAPE

you should then push the play button on your datasette recorder.

When the program has been found the screen will display

FOUND program name

on some Commodore computers you may then have to press the C=(Commodore symbol) key to then load the program. On other Commodore machines the program will load automatically. Check your owner's manual for specific loading procedures.

When the program has finished loading you will see the READY prompt and the flashing cursor beneath. Type RUN and press the RETURN key to start the program.

NOTES:

You should use the entire program name as listed to avoid loading programs that have similar titles.

Make sure that if you are loading VIC-20 programs you have the correct memory expansion cartridge (or no cartridge if that is required) plugged in before loading the program.

Some programs are divided into two sections, the main section (the one you should load first) and the MODULE section that is either automatically loaded when the first section is run or is loaded manually after the first section is run.

Commodore 64 programs will NOT normally run on a Commodore VIC-20 and by the same token VIC-20 programs will NOT usually run on a Commodore 64. Even though you may be able to load a particular program into the wrong computer it is unlikely that it will run properly.

ALWAYS refer to the article in the booklet for operating instructions, memory requirements, etc.



Sprite Control

BY M.J. CLIFFORD

Get a firm grip on your sprites with this program that lets you manipulate them quickly and easily.

There are many articles and programs for the Commodore 64 that deal with sprites. The purpose of most of these is to make it easy for you to design them. However, once you've designed the sprites, they still can be difficult to handle in Basic.

You must keep track of eight different pointer locations, eight y-coordinate registers and nine x-coordinate registers, as well as the various color and priority registers. The ninth x-coordinate register and the register that turns the sprites on and off require you to handle a single bit at a time. Bit manipulation requires that you know a little about binary numbering and the logical AND and OR operations. This can be a lot to handle even for the expert, never mind the novice.

Command Your Program

The program in Listing 1 adds to Basic four commands that take care of all these details. The commands turn sprites on or off, move them, automatically handle the problem of x being greater than 255 and put the pointers, colors, expansion and priority information into the proper registers.

The new commands are added to Basic by means of the SYS command, which transfers control to a machine language routine. The ML routine then reads the required values from the Basic program. Once the variable SP is set equal to 40080, the following commands are available:

Define a sprite: SYSSP,D,
 n,l,m,(c1,c2),
 c3,x,e,ye,p

Show a sprite: SYSSP,S,n

Hide a sprite: SYSSP,H,n

Move a sprite: SYSSP,M,n,x,y

Basic never sees the D, S, H and M subcommands, so you may use these letters elsewhere in the program. Of course, SP must not be changed from its value of 40080.

Run it
RIGHT

Commodore 64

The lowercase letters, which may be constants, variables or expressions, represent the following.

n—sprite number 0-7
l—sprite data location 0-255 (13-15 are in the cassette buffer and 11 is unused memory below the screen). The actual memory address is $1*64$
m—0 = monochrome, 1 = multicolor
c1,c2—the 01 and 11 colors shared by multicolor sprites (omit if m = 0)
c3—sprite color (10 colors for multicolor)
xe,ye—0 = normal, 1 = expand in x or y direction
p—priority 0 = sprite over text; 1 = text over sprite
x—x-coordinate 0-511 (24-343 are visible, others are all or partly off screen)
y—y-coordinate 0-255 (50-250 are visible).

The program in Listing 1 Pokes the machine language into memory and then saves it in a program file on disk.

Whenever you wish to use this program, you should load it by entering:

```
LOAD"SPRITECD",8,1 (1,1 for tape)
POKE 52,156:POKE 56,156:NEW
```

Any program using these commands should begin with the following lines.

```
10 POKE 52,156:POKE 56,156:CLR
20 IF PEEK(40080)<>32 THEN
LOAD"SPRITE CD",8,1
30 SP = 40080
```

Listing 2 is a short demonstration of how you may use these commands. It also includes a method for menu se-

lection by means of a mouse that is much cuter than the one that rolls around on the tabletop.

Description of Loader

10-30—Protect the machine language from Basic and set the beginning address.
35-60—Read the data from lines 1001-1019. The last number on each data line is a checksum. If you make a mistake in typing the data, the program will halt and report the line number of the data on which the error occurred.
100-110—report the successful loading of the package.
200-320—use the Kernal Save command to save the machine language in a program file that can be loaded directly with LOAD"SPRITECD",8,1 (1,1 for cassette—also change the 8 in line 260 to a 1).

Description of Demonstration

5-20—Protect ML from Basic, load the ML if necessary, and set SP = 40080.
25—Puts data for a striped-box sprite in location 13.
27—Reads data from lines 1010-1025 for the mouse sprite and Pokes it to location 14.
30-120—Move 8 sprites at once.
40-60—Use a loop to define 8 sprites.
50—Defines sprite Z in location 13, multicolor, colors 1, 7 and Z, not expanded in either X or Y and having priority over text.

55—Moves sprite Z to position X,Y (170,140) and shows sprite Z; only sprite 0 will be seen at first, since the other seven sprites are behind it.

70-120—Move all eight sprites away from the center, each in a different direction:

0 moves right as the X coordinate is increased by Z

1 moves right and up as X increases and Y decreases

2 moves up as Y decreases

3 moves left and up as both X and Y decrease

4 moves left as X decreases

5 moves left and down as X decreases and Y increases

6 moves down as Y increases

7 moves right and down as both X and Y increase

130—Hides all 8 sprites.

140—Moves sprite 2 to the upper left corner and shows it.

150—Moves sprite 2 diagonally across the screen from upper left to lower center.

160—Hides sprite 2.

170-230—Demonstrate expansion and color changes.

170—Puts sprite 6 into the center of the screen and shows it.

180—Redefines sprite 6 in location 13 as monochrome with color Z and expanded width.

190, 210, 225—Unexpanded.

200—Expanded height.

220—Expanded in both height and width.

230—Does it again in three more colors, then hides it.

300-385—Use a mouse to make selections from a menu.

300—Defines sprite 0 as in location 14, monochrome, color 15, unexpanded and without priority over text.

310-340—Print a menu.

345—Sets Y coordinate of sprite according to the value of C.

350—Moves sprite to location 24,Y.

355—Waits for keypress.

360—If f3 is pressed, increases C.

65—If f1 is pressed, decreases C.

370—If the return key is not pressed, goes to 345.

385—If the return key is pressed, C is the value of the option chosen.

1000—Time-delay subroutine. 

Address all author correspondence to M.J. Clifford, 2323 W. Bancroft St., Toledo, OH 43607.



Lister Filter

BY ALEJANDRO A. KAPAUAN

Convert those curious Commodore custom characters into something we humans can understand, and you will get easy-to-read, professional-looking listings.

A listing-translator program, Lister-Filter, allows you to print easy-to-read listings similar to the ones in this publication. It filters out all Commodore graphics characters output to the screen, printer or RS-232 devices (device numbers 3, 4 and 2, respectively) and replaces them with easy-to-read non-graphic equivalents.

For example, the clear-screen character will print as [CLR] instead of as a reverse-video heart character. A shifted space will print as [SHIFT SPC], while the character you produce by holding down the Com-

modore logo key and typing A will print as [COMD A]. In addition to these features, Lister-Filter also compresses long repeated sequences of graphics, cursor control or blank characters into a single string.

A string of 22 cursor-right characters will print as [22 RIGHT]. A single space will print as a space, while two or more spaces in sequence will print as [n SPC], where n is the number of spaces. The program is especially useful for making printed listings if your printer or printer interface has no graphics capabilities. Even if you do have a graphics printer, listings processed by this translator are more readable than regular graphics listings.

Lister-Filter was originally written for the expanded VIC-20, but it will run without modifications on a Commodore 64. It is written entirely in machine language and uses 630 bytes of your RAM. However, you don't have to know machine language to use the program and you do not even need a machine language monitor to type it in. The Basic loader program (see Listing 1) will do the proper loading and relocation of the Lister-Filter program.

**Run it
RIGHT**

VIC-20
Commodore 64

Using the Program

Type in the Basic loader program carefully. This may be a little difficult because of the numerous Data statements; however, checks are provided in the program so that you can easily locate the errors in the data when you run it. After typing in the program, make sure to save it.

When you run the program for the first time, it may contain some errors. If there is a Syntax error, examine and correct the offending line. If you get a message DATA ERROR NEAR LINE n, examine line n for errors, or possibly the line before it. If you get an Out of Data error, it is likely that you just left out a Data statement. Make the necessary corrections and save the program again.

Once the loader program executes to completion, and the screen displays FILTER INSTALLED, then all output directed to the screen, printer or RS-232 device will be translated. You may type NEW to delete the loader program, but the Lister-Filter program will still be there. To make a listing of another program, just load it into your computer and list it in the normal manner. To disable the translator when you no longer need it, you can warm start your computer by holding down the run/stop key and hitting the restore key. To reinstall the filter later, just load the loader program and run it.

You can modify what the translator will print for some graphics characters by chang-

ing lines 1000 to 1350 in the loader program. The variable C is set to the CHR\$ code of the character, while the string C\$ is set to the string to be printed. A GOSUB4000 then installs the code in the filter's exception table.

Lines may be added after line 1350 for other characters, like the extra Commodore 64 color-control characters, which are unavailable on the VIC-20. However, the total exception-table space may not exceed 256 bytes. The program will tell you if the table strings are too long. You might have to shorten some strings to make room for the others.

How It Works

Lister-Filter is basically a program that is placed just before the VIC's or C-64's normal output routine. This is done by modifying the output vector at locations 806 and 807 to point to the Lister-Filter program. After it does its translating, the program then passes control to the normal output program to print the translated characters.

The translating process is fairly straightforward. First, the character to be printed is checked against the previously received character. If it is the same, then a count of accumulated characters is incremented, and the character is not immediately printed. If it is not the same, then the character is saved, and the previously-buffered characters are printed.

That way, repeated characters can be compressed as a single string.

The character is checked to see if it is in the exception table. If not, it's checked to determine if it is a Commodore logo key graphics character. If it's neither of those, it's checked to see if it's a shifted character. The appropriate string representation of the character is printed, with any necessary numeric count. If the character is a normal printable one, then it's printed as is.

If you are knowledgeable in machine language programming, you might want to disassemble Lister-Filter to examine it in detail.

Lister-Filter is a handy program for making clear, professional listings, and it helps prevent the eyestrain and headaches caused by reading cryptic graphics characters. 

Address all author correspondence to Alejandro A. Kapauan, 141-6 Airport Road, West Lafayette, IN 47906.



Line Squeezer

Formerly "Compactor."

BY ROBERT W. BAKER

Here's a handy utility that allows you to squeeze your programs and thus gain more memory.

This was originally written for the Commodore PET, but I've made several improvements and have updated the program to run on the Commodore 64 and VIC-20 (with 16K expansion memory).

Compactor II reads a Basic program that has been saved on disk and creates a new, compacted copy. Compactor II deletes all REM statements, unnecessary spaces and leading colons.

This program, however, goes one step further. Whenever possible, it combines program lines to eliminate the link, line number and line-end-flag overheads normally associated with

each Basic program line. It makes a program as small as possible and usually faster running.

To give you an idea of what it can do, the Compactor II program is over 3100 bytes long (13 blocks on disk), and when compacted by itself, the new version is about 1800 bytes (8 blocks on disk), or approximately 58% smaller. Admittedly, the Compactor II program does contain a large number of spaces and several remarks, but the savings you get on any particular program depend on how it was written.

While writing the original version of this program, I came across a few of Commodore Basic's undocumented quirks. Since many of you like to experiment with the capabilities of having programs write other programs to disk, the following information may be of interest to you.

Zero-Length and Long Lines

Normally, it is impossible to create a zero-length line when you use the screen editor on any Commodore system. By zero-length line, I mean a line with a link, line number and end-of-line flag, but with no Basic commands or text. If you were to type just a line number using the screen editor, you would actually delete a line in-

**Run it
RIGHT**

Commodore 64
VIC-20 with 16K expansion

stead of entering a zero-length line.

When you write a Basic program on disk as a datafile, there is nothing to prevent you from entering a zero-length line. Basic, however, cannot correctly link the program lines when a program contains a zero-length line. Therefore, if you want the program to run, you cannot have any zero-length lines in your program.

At the other extreme, when you use the Commodore screen editor, you cannot create a Basic line that is longer than 255 bytes. The Commodore 64 normally limits you to a maximum of 78 characters, because of the line-wrapping characteristics and the need for at least a one-digit line number.

When you're writing a datafile to disk, be careful not to create a line greater than 255 bytes, as the program usually won't load back from the disk. If it does load, the program will normally be completely destroyed.

How the Program Works

When you run the Compactor II program, you have some control over the size of the program lines. The first input prompt (lines 410-420) will ask for the maximum line length. This must be a positive number between 1 and 255—the default value is 255 if you just hit the return key. When entering small numbers, be sure to use spaces to remove unwanted digits from the default number.

After you select the maximum line length, you are asked the name of the Basic program

file you want compacted (lines 440-450). (Remember that the program must be on disk.) If the file is not found or if any disk errors are encountered, they will be reported and the program will abort.

Next, you're asked to enter the name of the compacted program to be created (lines 460-470). This name cannot be the same as the original program name or any other file currently on the disk. If any file with the same name already exists, or if any disk errors are encountered, the program will likewise abort.

Compactor II reads the program as a sequential disk datafile, and the file is read twice. The first pass (lines 480-820) checks for line numbers that are the targets of GOTO, GOSUB, Run, or If. . . Then statements (lines 590-690). When it finds a target line number, that number is saved in matrix TL, if it's not already saved (lines 730-760). It also checks for multiple target lines in ON . . . GOTO and ON . . . GOSUB statements (lines 800-820).

As the first pass progresses, each target line is displayed in the order found (line 750). This gives you some indication of the scanning progress, since it can be rather slow. At the end of the first pass, the target lines are sorted into numerical order, to help speed up later processing (lines 860-890).

During the second pass (lines 930-1420), each line is copied, deleted or compacted according to the Compactor's rules. Again, the line number is

displayed as each line of the original program is processed, to let you know how the program is progressing. The rules followed by the Compactor are as follows:

- Any leading colons and/or spaces on a line are deleted (line 1010).
- A line that has only remarks is deleted if it is not a target line (lines 1020-1040). If the line is a target line, the remark will be replaced with a single colon, which must be retained (line 1050). This may produce a leading colon if the next line is not a target line and is combined with this line. The line cannot be reduced to a zero-length line, since Basic cannot link a program correctly with a zero-length line, as mentioned earlier.
- If any line contains a GOTO, Run or If . . . Then statement, it cannot be combined with another line. Line 1130 makes the check for these tokens and sets a flag in variable F whenever one is found. This flag forces the current line to be written to disk and the next line to be read without combining the two. Any line combined after these Basic commands would never be executed; thus the compacted program would not function properly.
- Any spaces within a line, that are not enclosed in quotes, are deleted (line 1110).
- Any remarks at the end of a Basic line are deleted to the end of the line (line 1140).
- Anything within quotes is copied untouched (lines 1180-

1200). If an ending quote is missing from a line that could be combined with another line, then an ending quote is added (lines 1210-1220).

- When a colon is found within a Basic line and not within quotes, the Compactor program checks the next non-space character before it copies the colon (line 1260). If a remark follows the colon, the colon and the rest of the line are deleted. Otherwise, the colon is copied, and processing continues as normal (line 1270).

- At the end of each Basic line, the Compactor checks to see if the next line can be combined with this one (line 1310). If there aren't any GOTO, Run or If . . . Then commands, and if the next line is not a target line, the lines are combined (lines 1320-1360). When this happens, the link and line number are discarded, a colon is written, and the next line is processed as part of the previous line.

- If the second line cannot be combined with the first line, the first line is written to disk with the correct link. This is the major difference between the original Compactor and this new version. Compactor II uses a line buffer to construct the entire line before it is actually written to disk. This allows the program to construct an accurate link value, which it will write at the front of each line.

- When the end of the program is found, the last line is written to disk, along with the ending zero link, and all files are properly closed (lines 1400-1420).

Lines On Lines

I skipped over the subroutines, which are located near the front of the program at lines 230-360. A GOSUB to line 240 will read a byte and return the character code established in the variable V, while starting at line 230 will read two bytes. Lines 270-290 check for disk errors and report any findings before aborting the program.

Lines 300-320 compute the link value for a program line in L\$ and then write to disk the link, program line and ending flag (0). Lines 330-360 read an entire Basic program line, saving each byte in the C matrix and the line length in PL. It also computes the program line number, saves it in the LN variable and displays the line number, overwriting any previous number.

The line length that you specify at the program's beginning prevents Compactor II from combining lines that would exceed the specified limit. However, any lines already greater than the limit

will be copied without being combined with other lines. If you select a small limit, then the program will copy most of the lines without combining them. It will, however, compact each line by removing spaces, remarks or leading colons.

Keep in mind that the newly compacted program may have lines that cannot be edited with the screen editor. On the C-64, any program line that exceeds two screen lines cannot be edited. (See my Uncompactor program, following, which allows you to break and shorten program lines to allow for easier editing.)

One quick word of caution. When you enter the Print# command, do *not* use the Print statement's abbreviation, which is a question mark. If you do, the line will still list correctly, but internally the code is incorrect and will generate a syntax error. Always type the entire command—PRINT#. □

Address all author correspondence to Robert W. Baker, 15 Windsor Drive, Atco, NJ 08004.



Line Expander

Formerly "Uncompactor."

BY ROBERT W. BAKER

Are your program lines overcrowded with multiple statements? This program breaks up and shortens those lines, making them easier for you to edit.

This was originally written for the Commodore PET, but I've made several improvements, including updating the program to run on the Commodore 64 and VIC-20.

Uncompactor, a companion to Compactor II, reads a Basic program that was saved on disk and creates a new, uncompacted, or expanded copy. It does this by taking any multi-statement lines (statements separated by colons) and breaking them into separate program lines with new line numbers.

**Run it
RIGHT**

Commodore 64
VIC-20 with 16K expansion

Long lines created by Compactor II can now be edited and the program recompacted.

When you split multistatement lines, the new line numbers are created by incrementing the original line numbers by 1. I follow this procedure until a line number reaches the next original line number in the program. When I reach that point, I then copy the remainder of the original line as part of the last line generated, with any appropriate separating colons.

The program must take into account certain Basic tokens, or keywords, since they determine whether or not a particular line can be broken into separate lines. Thus, any data following a GOTO, End, Run, If, Return, REM, Stop, List or CONT token is copied, unchanged, to the end of the current program line. Also, once a program detects a quote, it must copy the line until it reaches another quote or the end of the program line.

Program Description

When you run the Uncompactor II program, you have some control over what size program lines will be uncompacted. The first input prompt (line 370) will ask for the minimum line length to try uncompacting. This should be a positive number between 1 and

255, but there is no check of the value entered. If you just hit the return key when prompted, the default will force the program to try and break every single line, where possible. Selecting a number like 20 will cause small lines (with 20 or fewer characters) to be left untouched, while longer lines are uncompactd.

After you select the minimum line length, you're then asked the name of the Basic program file to be uncompactd (lines 390-400). The program must be on disk, as program files cannot be read from tape. If the file is not found, or if any disk errors are encountered, the program will terminate.

Next, you're asked to enter the desired name of the uncompactd program to be created (lines 410-420). This name cannot be the same as the original program name or that of any other file currently on the disk. If any file with the same name already exists, or if any disk errors are encountered, they will be reported, and the program will terminate.

Uncompactd II treats the program to be uncompactd as a sequential disk data file, which it reads only once. As it reads the original program, each line number is displayed on the screen (lines 490-510). This gives you some indication of how things are progressing as Uncompactd II runs; it can be rather slow.

After Uncompactd II copies the original line number, it reads the actual line into the C matrix (lines 550-560) and then

reads the next link and line number (lines 600-620). When it finds the zero link at the end of the program, the next line number is forced to 64000. This number exceeds any possible Basic program line number, thus forcing proper handling of the last line of the program read.

Once the entire line has been read, and if it's longer than the limit you selected, it is scanned for colons and certain Basic tokens (lines 660-940). If the line is shorter than the specified limit, it's copied untouched (line 860). If one of the special Basic tokens is found (lines 820-850), the remainder of the line is copied untouched.

When a colon is found, the line is split as long as the current line number plus one is less than the next original line number (lines 700-760). The current line is written to disk with the proper link and ending flag. Single leading colons at the start of any line are retained, while spaces or extra colons following any colon in the middle of a line are deleted (line 750).

Whenever quotation marks are encountered, the remainder of the line is copied untouched until the quote closes or end of line is found (lines 910-940). At the end of the program, a zero link is written to disk to properly terminate the new program, and all files are closed.

The subroutines are near the front of the program to help speed things up. The subroutine at line 230 reads two bytes, while 240 reads a single byte

from the original program file. When the program returns from this subroutine, the last character read is in C\$, with the character codes in V and V1. Lines 270-290 check for disk errors and either return to the calling line or display the disk error and stop the program. Lines 300-320 calculate the correct link for the line in L\$ and write the entire line, along with the link, to the new program file.

Newly created, uncompactd programs are fully linked and ready to run. You can use the Uncompactor II and Compactor II on any standard Basic pro-

gram that does not contain embedded assembly language routines.

As I mentioned in the Compactor II article, don't use a question mark as an abbreviation for Print in Print# commands. The line will still list correctly, but internally the code is incorrect and will generate a Syntax error if executed. Be sure you always type the entire command—PRINT#—to avoid problems. ☐

Address all author correspondence to Robert W. Baker, 15 Windsor Drive, Atco, NJ 08004.



Datafile

BY MIKE KONSHAK

If you want to computerize all those records you have to keep track of, here's a dandy database program that will give you information in a jiffy. (The first of two parts.)

A database is one of the more practical and useful programs that a computer owner can have. Information storage has always been one of the major areas of emphasis in the computer industry, and for the personal computerist there are many applications, particularly in maintaining files such as family records, Christmas card lists, recipe files, inventories of personal possessions and anything else you might want to keep track of in a handy and organized manner.

Databases take many forms and may be programmed in several ways, according to your needs and the extent of the data to be organized and stored. Databases normally require some type of mass storage device, such as a tape or disk drive that will keep the rec-

ords for later processing.

Printers are also a major peripheral used with databases. They produce the hardcopy reports without which the accumulation of records would be extremely hard to analyze. After all, what good is the computer without output?

Relative Records

A database is essentially a program that creates a program that collects and processes records according to your wishes and needs. It consists of records and fields. A record is basically a collection of information in the form of fields, each one containing information unique to that record. All records in a particular database have the same number of fields containing the same types of information.

For example, consider the following database, containing a list of family members and friends with their birthdays and gift preferences:

DATABASE:Birthdays

1. Name: Mike K.
Born: 05-28-47
Likes: Computers
2. Name: Becky K.
Born: 06-27-58
Likes: Clothes
3. Name: Sarah K.
Born: 09-10-75
Likes: Drawing
4. Name: George S.
Born: 07-03-50
Likes: Wine

**Run it
RIGHT**

Any ASCII or Commodore
printer
Commodore 64 with 1541 disk
drive

5. Name: Leslie Z.
Born: 01-18-43
Likes: Books

In this database, called Birthdays, there are currently five records. Each record contains three fields, entitled Name, Born and Likes. The information in each field is the actual data that's being recorded and organized. Such data is sometimes called an item.

As you can see, the data is not listed in any particular sequence or order. One of the features of a database program is the ability to manipulate or sort records in alphanumeric sequence, according to a particular field. Obvious sorts would be into lists by name or by birthday. In this way you'd be able to print out a list of records in any convenient order.

For example, if a sort were performed on the first field (Name), the order of the records would be 2-4-5-1-3. Notice that the sort was keyed from the first name, not the initial of the last name. Sorts always start with the leftmost words and characters. If the key field was Born, the order would be 5-1-2-4-3. Notice that this sort is determined by the first numeric character, which happens to be the month, not the year.

Main Database Features

Features found in most databases include:

- ADD additional records.
- MODIFY existing records.
- DELETE records from database.

- SORT records by field.
- LIST records on the screen.
- PRINT list of records on the printer.
- SEARCH for one or more similar records.

When the Print feature is chosen, you typically have the ability to format the list of records in various forms and arrangements. Mailing labels and reports are examples of common uses. Reports typically have headings at the top of the page, with the data items listed in columns below the headings.

Not every field of a record needs to be printed if the information in some fields is not useful for the report. You normally design the format for a particular report or type of mailing label and save it on disk for later use. You usually store formats separately from the information in the database.

The records in a database are normally stored on disk. Tape drives are also used, but are very slow, especially when dealing with a large number of records. Tape drives always store the database in the form of a sequential file, while disk-based systems can store the records in either a sequential or a relative file.

Sequential files involve loading the entire database from the disk into the computer's memory. You then manipulate the records, and the information is printed out while the memory contains the database. When you've completed all desired operations, the updated database is saved back to the

disk. You normally scratch the old information before saving the new.

Sequential files loaded into memory allow very fast operations on the data. The major drawback is that the computer's memory capacity limits the size of the database or the number of records. It's very important to keep the number of fields in a record small, as well as to keep the length of the items in each field to a minimum. This will allow the maximum number of records.

In contrast to sequential files, relative files store data in specified disk areas called sectors. You can access and manipulate each record without affecting other records. Since you can only perform operations directly on the disk, instead of in memory, these systems may be very slow, especially when you perform sorting operations.

Printing operations pull the data directly from the disk one record at a time. This also takes longer than it would with a memory-based system. The main advantage of relative files is that the database can store over three times as many records as a memory system.

Another advantage is that you can develop a more complex and extensive database program, since memory space is not needed for loading records. Relative file systems can add features such as mathematical calculations of records, graphics routines to plot out data and screen formatting

to aid in entering data.

Datafile Description

Datafile is a memory-based, multiple-program database system for the Commodore 64. It utilizes sequential files on a 1541 disk drive, and any ASCII or Commodore printer. I chose a memory-controlled system because most home users of personal computers don't need a large number of records. Also, you could probably better use the time spent in front of the computer for more recreational activities, like programming, than in waiting for a disk-based system to perform, especially when using the slow 1541.

Datafile allows you to create your own database, choosing the number and length of fields, as well as their titles. The program will calculate the maximum number of records that can be retained in memory according to the criteria you established. After you've created a database and added records, you can perform standard operations on the data and save to disk or print out the results in various formats.

Datafile uses several techniques to save time and memory space. The main Basic program, Datafile, is typically loaded first at the beginning of RAM. When run, the program establishes the existence of every variable that Datafile and its subprograms will use, by setting each variable to a dummy value.

String variables are set to a null [A\$ = CHR\$(0)] or some value pertinent to the program, and floating point and integer variables are assigned a value of zero [A = 0]. Finally, whether a file of data is created at the start of the program or by loading an existing file, the arrays are dimensioned last. This has a two-fold purpose. It allows programs to load other programs and also minimizes the time the computer has to spend managing the memory.

The main program can load other subprograms from the disk, removing itself from memory to allow room for the new program in the same memory cells. The new program will then be able to use the same variable values and data which were set and retained during the operation of the first program.

This only works as long as the second program is smaller in memory requirements than the first. The second program, however, can load the *first* one, even though the first is larger, because the memory space was allocated when Datafile first was loaded.

Garbage Collection Time

The Commodore architecture is such that as the program encounters variables during execution and, except for strings, gives them values, those values are stored directly above the Basic program. When an array is dimensioned, the computer will assign the empty cells directly above the variable values to array data.

Consequently, as the computer encounters a new variable that it has not seen before, it will start shoving the array higher up in memory, cell by cell, until enough room is available for the new variable. (Evidently, variables get lonely if they're not together.)

After the arrays, strings are stored. Basic has a nasty habit of reappropriating memory space that's holding strings, in order to free up the memory for possible future needs. This procedure, called garbage collection, is normally invoked when the Basic statement FRE(0) is in use. Garbage collection takes time, especially when dealing with large arrays consisting of strings, such as those created by Datafile.

It's possible to lose control of your computer for several minutes when this happens, and it will occur every time another new variable comes along. I advise you to keep this in mind when programming with arrays. It's best to keep the number of variables to a minimum and to predefine them before dimensioning arrays.

As just mentioned, the program retains all the record data inside memory, even though programs are being removed and replaced with other programs. The subprograms perform operations and manipulate the record data as utilities serving the main program. If the routines and services provided by the subprograms DFReport and DFMail were combined with Datafile into one large pro-

gram, there would be less space left for records.

The Subprograms

The following is a brief description of the function of each of the subprograms.

(Note: For reasons of space, the subprograms DFReport and DFMail will appear next month, along with a detailed description of each.)

The Datafile main program creates the database, defining the number of fields per record, the titles and lengths of fields, and the number of possible records, based on how the fields were set up. Datafile also sorts the fields in alphanumerical order, depending on which field is chosen.

Datafile also acts as the controlling program for disk-related operations, such as loading (reading) and saving (writing) datafiles, formatting blank disks, reading the directory and choosing which subprograms to advance to.

The DFMail subprogram is designed to print mailing labels and has the capacity to determine which fields will be printed on which lines of the label. DFMail prints on any single-row, tractor-feed labels, and can adjust the number of lines per label and the number of characters per line.

Once the label's format is designed and saved for future use (in special format files), you have the ability to search through the datafiles for selected records with common fields (e.g., Name = Smith) or to print the entire datafile. In other

words, you can pick and choose as you like.

The DFReport subprogram is designed to print reports on the Commodore 1525E or MPS801 printer, as well as any ASCII-type printer with suitable interface. DFReport has been tested satisfactorily on Okidata, Epson and Gemini dot-matrix printers, as well as on the Brother daisy-wheel typewriter/printer. A Cardco interface was used with all of the above hardware.

You have the capability to format the report in order to present the records in the way best suited to your needs. You can save the format for recall when another report must be printed. You can print up to 136 characters across the page, depending on the capacity of your printer. The Commodore printers will only print reports up to an 80-character width.

A title consisting of four lines will be centered at the top of the page, followed by column headings. You can define up to eight columns with the width and location of each. You can also define the column titles, although these normally have the same names as the record fields that will be printed below the headings. You may then search selectively for the records to be printed in the columns.

For long reports, the printer will automatically number the page, advance to the next page, and print column headings before beginning to print more records. For a faster printout of the records, a nonformatted print utility is provided that lists

each record and every field within the record in rows instead of columns. This printout can be cut and pasted on 3 × 5 cards.

All the above programs save datafiles or format files under special names that can be loaded only by the program that saved the file. In many cases the name given to the original datafile when the database was created will also be used as a reference on format files created by DFReport and DFMail. This feature helps you keep track of which format went with which datafile. Consequently, Datafile, DFReport and DFMail could each have a file named Xmas Mail, but would load only their respective file.

Datafile Instructions

You begin by typing LOAD "DATAFILE",8 <RETURN>. When the disk drive stops running, type RUN <RETURN>.

The screen then displays the main menu, which resembles the following. (Text or letters surrounded by brackets denote reversed video characters, normally identifying a key to be pressed.)

[DATAFILE MENU]

[C]REATE NEW FILE
[A]DD RECORD TO CURRENT FILE
[M]ODIFY RECORD IN CURRENT FILE
[D]ELETE RECORD IN CURRENT FILE
[R]EAD OLD FILE FROM DISK
[P]RINT RECORDS BY SELECTION
[V]IEW FILE ON SCREEN
[S]ORT RECORDS BY FIELD
[W]RITE NEW FILE TO DISK
[F]ORMAT DISK
[S] DISK DIRECTORY
[Q]UIT PROGRAM
[PRESS THE APPROPRIATE KEY]

THERE ARE 0 RECORDS IN MEMORY SPACE FOR 0 MORE RECORDS

(Note: The last line will not be displayed until a file has been created or loaded from disk.)

You can choose any of the 12 options by pressing the key that represents the first letter of the option, although Create or Read should be the first one chosen when you begin. The program will jump to the respective subroutine without your having to press the return key. When a particular subroutine has completed its chores, it will always return to this menu.

It's a good idea to create a small database at first, in order to become familiar with Datafile. Don't put too much effort into the first go-around. Experiment a bit to check out the program's capabilities. The following is a step-by-step description of what to expect when you select options from the main menu.

Create New File

Try to maximize the available memory space by keeping the number of fields and the lengths of the names to a minimum. The lengths of the fields should always be restricted to less than 75 characters.

Here's a practice file, which we'll call Names and Ages. It will have two fields, the first one to be called Name, and the second one Age. We'll only be putting first names in our database, so a length of ten characters for field 1 should be adequate. We will be putting

the person's age in field 2, so two characters should suffice. Press the return key after every prompt. The program continues with:

HOW MANY FIELDS IN EACH
RECORD? 2

FIELD#1
TITLE? NAME
LENGTH? 10

FIELD#2
TITLE? AGE
LENGTH? 2

The computer will then calculate as closely as possible the number of records that can be stored:

YOUR SELECTIONS WILL ALLOW 1110
MAX RECORDS. [A]CCEPT OR
[R]EJECT?

Press A. If R is pressed, the program will return to the point where you are asked for the number of fields in each record. This gives you the chance to change the fields in case you didn't get as many records as you were expecting.

If you press A, the main menu should appear, and the bottom line should tell you again how many records the memory can hold. This will decrease by one every time you add a new record.

Add Record to Current File

After pressing A on the main menu, the screen displays:

PRESS THE [RETURN] KEY AFTER
EACH ENTRY
PRESS [RETURN] WITHOUT ANY
ENTRY TO STOP

[RECORD NUMBER 1]

NAME? MIKE
AGE? 36

Now type in about ten records so you'll have something to play with. If you try to enter into any field more characters than that field was initialized for, you'll receive an error message. You will notice the dummy character behind each input statement. This is used to reserve the space while the computer is writing the sequential file to the disk.

You stop adding records by pressing the return key without making an entry in the first field. This doesn't work on succeeding fields because it's assumed there's some data there that needs to be saved. This also allows you to fill in blanks later if information is unknown at the time.

It's important to note that Datafile uses Input statements that do not allow the use of quotation marks, commas, semicolons or colons as part of data in the fields. All other alphanumeric characters are acceptable.

Modify Record in Current File

If you press M, you will see:
MODIFY WHICH RECORD? ENTER [#]
OR [A]LL
?

If you want to change just one particular record, enter the number of the record (try 1), then press the return key. Pressing A will display all the records in the file one at a time. Pressing 1 brings this to the screen:

TO MODIFY RECORD NUMBER 1,
MAKE CHANGES AS EACH FIELD IS
DISPLAYED, THEN [RETURN]

NAME? MIKE
AGE? 36

As you can see, this format is similar to the Add operation, except that the data is pre-printed for you on the screen. Press the return key once, accepting the name, then update the age by typing over the 36 with a 37; press the return key.

The main menu should reappear again. If the entire file is going to be modified, holding the return key down will scroll through the data. It is best, though, to find the record you want with the View function.

Delete Record in Current File

Pressing D gives you:

DELETE WHICH RECORD? ENTER [#]
OR [A]LL
?

Don't be afraid to press A on this one. Records will not be deleted unless you've given the go-ahead to do so first. For this example, enter 1, then press the return key, which displays:

TO DELETE RECORD NUMBER 1,
PRESS [SHIFT] [D], PRESS [SPACE
BAR] TO ADVANCE

The entire record is displayed so that you'll be aware of the total contents of the record before you try to delete it. If you want to delete the record, hold the shift key down while you press the D key.

The total number of records in the file will be decreased by one, and all the records after the one you deleted will be re-numbered accordingly. If you decide not to delete the record after all, just press the space bar and it will advance you to

the next record or bring you back to the main menu. Remember to save your revised file.

Read Old File from Disk

This utility is normally performed at the start of Datafile to load a previously stored file. The program prompts with:

ENTER NAME OF FILE TO BE LOADED
?

Type in the name of the datafile and press the return key. The file will load and you'll return to the main program. If the return key is pressed without a filename present, the program will also exit safely back to the main program.

Print Records by Selection

This utility advances you to another menu, designed to load subprograms that will actually perform the printing operations. If no records are present in memory, you'll be directed back to the main menu.

[PRINTER MAIN MENU]

PRINT RECORDS USING:

[R]EPORTS AND LISTS
[M]AILING LABELS
[U]SER DEFINED SUBPROGRAM
[E]XIT TO MAIN MENU
[PRESS THE APPROPRIATE KEY]

E returns the program to the main menu, R loads up DFReport and M will load DFMail. Pressing U results in:

ENTER NAME OF SUBPROGRAM
?

Here you can load up programs that might perform other operations on your data that Datafile does not provide. Not

entering any filename at all will get you back to the printer menu.

User-Defined Subprogram serves to load a program that you might write to enhance your particular datafile. One example might be a program that adds up all the numeric values in one field of a datafile. This could, for example, be a field that holds the current value for household inventory items, giving you quickly the total value for insurance purposes.

Many variations are possible. You will have to study the program listings to find the variable names required to get the correct data. More on this later.

View File on Screen

Entering this routine displays the first record in the datafile with the following commands below the record:

[RECORD NUMBER 1] IN FILE (name of datafile)

(Record Data)

[N]EXT, [L]AST, [J]UMP, [F]IND, [E]XIT

Pressing N causes the screen to step to the next record. You can walk through your entire datafile, one record at a time, up to your end record with this command. L steps you backwards, decrementing each record number by one, to previous records.

J allows you to jump directly to a particular record number, instead of stepping one by one. You'll be asked for the record number; then enter your choice and press the return key.

F is a search function that

allows you to find record fields that share common items or data. The screen displays a list of the field names of the current data file, then asks you to enter the number of the field you wish to search. The field name is then displayed and you're asked to ENTER [COMMON ITEM]. Enter the string of text that is to be searched and press the return key.

For example, if you chose a field which was named First Name, you might enter the string JIM. The computer will search out all records that begin with JIM in the First Name field.

Not only would JIM come up, but JIMMY would also be displayed because it begins with JIM. Entering A would cause a search of all strings in a particular field that began with A and so on. Press N to continue to the next record in the search.

Sort Records by Field

The field names will be displayed, each preceded by a number, and the list will be followed by a prompt:

WHICH FIELD IS TO BE SORTED?

Entering one of the numbers shown, followed by RETURN, will send the computer off to sort out that particular field in ascending alphanumerical order. The computer will tell you how it's doing during the process by flashing the number of the record it's currently working on.

Datafile uses a Shell-Metzler sort routine. All the data items

entered into Datafile are stored as string values in the arrays, whether the value is in the form of alphabetical characters or numbers. Therefore, here is a point to consider in the sorting of string variables that are numbers: The first number encountered will be considered the first character used for comparing against another number.

Given the numbers 2000, 35 and 156, the sort routine will compare the 2 in 2000 with the 3 in 35 and the 1 in 156. The result will be shown as the sequence 156, 2000 and 35.

This is obviously not the intention. You can get around this problem by entering numbers that have the same number of digits. The numbers will now look like this: 2000, 0035 and 0156, and when sorted, will be in the proper order: 0035, 0156, 2000.

All the records will now be in a different order according to the chosen field. If you want to keep the file in this order you must write it back onto the disk.

(Note: Any desired sorting should be done before advancing to the Print subprograms. There is no facility for sorting the records in those programs.)

Write New File to Disk

Entering this routine produces this display:

```
ENTER NAME OF CURRENT FILE TO  
BE SAVED (12 CHARACTERS MAX).  
ANY EXISTING FILE WITH THE  
SAME NAME WILL BE SCRATCHED  
?
```

As mentioned previously,

Datafile adds special character codes to the beginning of your datafiles and format files. This ensures that programs will load their own files and allow the multiple use of the same filenames.

Writing your current datafile onto the disk invokes the following operations:

- Datafile Mail List was read into memory from the disk. It appears on the disk directory as DFJ MAIL LIST.
- The current file has been updated and the file is entered, for writing the file to the disk, exactly as the name that it was read from, MAIL LIST.
- The program will change the name of the last file on the directory named DFJ MAIL LIST to DFJ MAIL LIS!OLD. Notice that the last four characters in the 16-character filename will be replaced with !OLD.
- The current updated file will then be saved as DFJ MAIL LIST.
- If DFJ MAIL LIS!OLD was already on the disk directory, that file would be scratched before the MAIL LIST file is renamed.

In essence, Datafile always keeps your current datafile as well as your last datafile. This gives you the opportunity to recapture the last version of your data. If you desire to load the last version from the Read Old File on the menu, enter MAIL LIS!OLD. Do *not* include the special characters shown at the beginning of the filename on the directory.

If for some reason you desire

to keep the !OLD files, you must give them a new name to keep from scratching them later. (Remember, do not exceed 12 characters.)

Format a Disk

This feature allows you to format a blank disk for use later on in saving files.

[DISK NAME,ID]?

Insert a disk into the disk drive. Enter up to a 16-character header for the disk name, followed by a comma, then a two-character disk ID, and finally a RETURN. The drive will begin to format the disk (this takes approximately 3½ min-

utes). When it is finished, you will be returned to the menu.

\$ Disk Directory

Pressing the shift and 4 keys will list the directory of the current disk in the drive. Press any key to get back to the menu.

Quit Program

This command makes a clean exit out of the program. It closes all the files, performs a housekeeping function and lets you know if you've forgotten to save your current file. Any modifications to a file will trigger a flag that will prevent you from immediately leaving the program.

Datafile Part II

DFMail Instructions

You load DFMail using the print options found in Datafile. It is assumed that a datafile is currently held in memory; otherwise, there will be nothing to print. The screen shows:

[LABEL SIZE]

[S]TANDARD—5 ROWS PER LABEL
15/16 BY 3½ INCHES

[L]ARGE—8 ROWS PER LABEL
17/16 BY 3½ INCHES

[O]THER—CUSTOM LABEL SIZE OR
NUMBER OF CHARACTERS PER ROW

NOTE: LABELS ARE SEPARATED BY
ONE ROW
32 CHARACTERS PER ROW IS
STANDARD

[PRESS THE APPROPRIATE KEY]

DFMail uses "One-up" tractor-feed labels and is adaptable to any length or width of label. The standard size labels (with 5 rows of text) are the most popular and most easily obtainable, with the large size (8 rows) being next in line. Press either the S or L key. If you have labels of a non-standard size, choose the O option instead.

OTHER is adaptable to let you choose the number of rows, from 1 to ?, and the number of characters can be expanded from the standard 32 up to 136. Putting your printer into compressed mode will allow more characters on labels of standard length.

Some labels that fit the non-tractor-feed printers give you two across the page. These labels, which are four inches long, are used if the printer has only pin feed (Okidata and Epson, for example). These longer labels can accommodate 38 characters per row, if desired. DFMail, however, will only print on the leftmost labels. You can, of course, feed the labels in backwards to use the other side.

If <O> is pressed, the screen will display:

```
ENTER NUMBER OF ROWS ON
      LABEL?
ENTER NUMBER OF CHARACTERS
      PER ROW?
```

Enter your modifications when prompted. The next screen shows the main menu for the mailing labels program, as follows:

[MAILING LABELS MENU]

```
[P]RE-DEFINED FORMAT OR
[D]EFINE NEW FORMAT
[C]HANGE LABEL SIZE
```

```
[E]XIT TO MAIN PROGRAM OR
[R]EPORT/LISTING PROGRAM
[Q]UIT PROGRAM
```

[PRESS THE APPROPRIATE KEY]

Pressing the E key reloads Datafile into memory without disturbing the record data. Q closes the files and terminates the entire program. Ending here wipes out all data. Do this only if you have not updated any records and if you have your current datafile stored on disk. You will be warned if you have not done so.

R loads the subprogram DFReport directly, instead of

having to go back to the Datafile program. C sends you back to the first screen that you encountered when you entered DFMail. This allows you to alter the size of your labels and printouts.

Formatting Your Labels

Formatting of printer outputs may be the most confusing aspect of a database. You must be able to visualize how you want the final result to appear. This may seem difficult at first, but being able to customize your outputs is considered a strong feature of a database.

Fortunately, once you have formatted a label or report (when using DFReport), you'll be able to save your design for future recall. From then on, when you want to print your labels, you'll be able to breeze by the formatting routines.

Let's design a sample mailing label that will probably meet most of your needs. Before doing this, you must have a previously created datafile that's compatible with your label format. The datafile will have the following structure:

Name of datafile: MAIL LIST
Number of fields: 8

Field #	Field name	Field length
1	LAST NAME	15
2	FIRST NAME	10
3	CODE	5
4	STREET	32
5	CITY	23
6	STATE	2
7	ZIP	5
8	PHONE	12

Modifications to the above datafile might include a second address line (e.g., COMPANY

NAME). The phone number is included in the datafile, but will not be printed on the labels. The field Code may be used for classifying the records (e.g., R=relatives, F= friends, B=business associates), or for an employee number, a professional title or an account number for business purposes.

Define New Format

Now that the datafile is defined, and assuming that records are present, let's return to where we left DFMail. Pressing D in the Mailing Labels menu results in this display, which will indicate, by rows and characters, which label size has been chosen:

[MAILING LIST FORMAT]

THIS FORMAT USES SINGLE ROW LABELS.

EACH LABEL CONTAINS UP TO 5 ROWS.

EACH ROW CAN CONSIST OF 1 TO 3 FIELDS.

IF THE LENGTH OF MULTIPLE ITEMS EXCEEDS 32 CHARACTERS, SOME DATA WILL BE CUT OFF.

[NUMBER OF ROWS?]

At this point, let's pause to discuss what your label will look like. Row 1 will include record fields 1, 2 and 3 (LAST NAME + FIRST NAME + CODE), in that order. Row 2 will only have record field 4 (STREET). Row 3 will consist of record fields 5, 6 and 7 (CITY + STATE + ZIP). Rows 4 and 5 will not be used.

The label shown on the screen is divided into 3 fields per row. These are *format* fields, not *record* fields. Try not to get them confused. Enter 3

for the NUMBER OF ROWS and press the return key.

CHOOSE WHICH FIELDS GO IN WHICH ROW
ENTER [0] IF ADDITIONAL FIELDS ARE NOT DESIRED.

1 LAST NAME ROW 1
2 FIRST NAME FIELD 1? 0
3 CODE
4 STREET
5 CITY
6 STATE
7 ZIP
8 PHONE

Field 1, in this case, refers to the first field or item of the first row. In this field we will place record field 2, which is displayed on the left of the screen. Respond to the prompts on the right of the screen as follows:

ROW 1 press the return key
FIELD 1? 2 after each entry
FIELD 2? 1
FIELD 3? 3

ROW 2
FIELD 1? 4
FIELD 2? 0
FIELD 3? 0

ROW 3
FIELD 1? 5
FIELD 2? 6
FIELD 3? 7

The screen will now display:

DO YOU WISH TO REVIEW YOUR FORMAT AND/OR MAKE CORRECTIONS?
[Y] OR [N]

Pressing Y will repeat the last screen, except that the record-field numbers will appear after the format-field prompts. Press N to advance into the program.

[SAVE FORMAT] [Y] OR [N]? Y

SAVE UNDER WHAT FILE NAME?
? MAIL TEST

You will notice that the pro-

gram pre-prints the filename that was determined when your datafile was saved or loaded during a disk operation. This links record and format files together so that you will not have to remember different names. At this point any format files with the name Mail List will be scratched as this new format is saved. Unlike the datafiles, format files will not be given a backup when a file of the same name is resaved after changes. Change the name of the format file at this time if you want to retain the old format, and press the return key.

The program will then advance to where the labels are aligned in the printer. Jump there now if you wish, because the next few paragraphs will discuss the situation where the user loads in a pre-defined format.

Pre-defined Format

After pressing P at the Mailing Labels menu, the screen will display:

```
LOAD FORMAT FROM WHAT FILE?  
? MAIL TEST
```

The prompt should pre-print the last-used filename. If MAIL TEST is the correct format file, press return. As soon as the file is loaded, the program will display:

```
DO YOU WISH TO REVIEW YOUR FOR-  
MAT AND/OR MAKE CORRECTIONS?  
[Y] OR [N]
```

This is the same question asked when you first designed the format. If you are not sure if the format you loaded was the correct one, you may check it

at this time. This is also a good opportunity to make a slight change for a one-of-a-kind job. Press N. The screen will display:

```
SAVE FORMAT? [Y] OR [N]
```

This may seem repetitive, but it allows you to save a changed format, or to save the current one under a new name, or on a new disk. Press N. The program will next display:

```
INSERT SINGLE ROW TRACTOR FEED  
LABELS  
RUN TEST LABELS TO HELP POSITION  
LABELS
```

```
PRESS [T]EST LABEL  
[C]HOOSE RECORDS
```

Pressing T will print rows of asterisks. The number of rows and characters should reflect your label size and format. Position the labels in your printer so that the rows appear centered in the label. Once the labels are aligned, press C to advance to the Print Options menu, where you will choose the records to print.

```
PRINT OPTIONS MENU
```

```
[A]LL RECORDS IN FILE  
[S]ELECT INDIVIDUAL RECORD  
[F]IND RECORDS WITH COMMON  
FIELDS  
[E]XIT TO MAIN MENU  
[PRESS THE APPROPRIATE KEY]
```

At this time, you actually decide which records you want to print, and then begin printing. (If, at any time, you decide that you want to leave this section—before or after printing—press E to get back to the main menu.) The choices are as follows:

All Records in File

The printer will start printing from record number 1 until it has printed your entire datafile. Sit back with a cup of coffee if you have a large file.

Select Individual Record

This gives you the opportunity to print just one label of your choice. This assists you in making last-minute corrections or printing just a few records out of your datafile. The screen displays PRINT WHICH RECORD? Enter the record number, then press return. If you type in a number higher than the size of your datafile, you will receive an error message. You must print *something* to get back to the menu.

Find Records with Common Fields

This search routine operates identically to the one in the view option of the Datafile program. The screen will display all the field names in your datafile to help you search. The following list is from the datafile called MAIL LIST. For this example, we will search for all last names beginning with S.

FIND RECORDS WITH COMMON FIELDS

- 1 LAST NAME
- 2 FIRST NAME
- 3 CODE
- 4 STREET
- 5 CITY
- 6 STATE
- 7 ZIP
- 8 PHONE

WHICH FIELD IS TO BE SEARCHED? 1

ENTER [COMMON ITEM]
(THE ENTIRE STRING IS NOT REQUIRED)

[LAST NAME] ? S

SEARCHING RECORD

If you followed the above sequence, the # symbol will be an incrementing number that will stop when the program finds a record with a last-name field beginning with S. It will then print out that record and then start looking for another. If you had previously sorted this file by last name, all the Ss would be printed one after another. The program will continue searching until it runs out of records. It will then send you back to the Print Options menu.

If you had typed in SWYKOWSKI for the last name, only those records that perfectly matched, or began with SWYKOWSKI, would be printed.

For a business application, you could use this feature to group mail by zip code. It is also possible to print only those records that have a special code that was previously entered in the code field of the record.

DFReport Instructions

Just as with DFMail, you load DFReport using the print options in Datafile, and again it's assumed that a datafile is in memory; otherwise, there is nothing to print. The screen shows:

[REPORT PRINTOUT MENU]

[L]IST RECORDS UNFORMATTED OR
[P]RE-DEFINED FORMAT
[D]EFINE NEW FORMAT

[E]XIT TO MAIN PROGRAM OR
[M]AILING LABEL PROGRAM
[Q]UIT PROGRAM

[PRESS THE APPROPRIATE KEY]

This menu functions like DFMail. Pressing E reloads Datafile back into memory for further updates without disturbing the record data. Q closes the files and terminates the entire program. Ending here wipes out all data. Do this only if you have not updated any records and if you have your current datafile stored on disk. You will be warned if you fail to do so. M loads the program DFMail directly without first having to load Datafile.

We'll be using the datafile MAIL LIST, as described in the DFMail instructions, as an example file to demonstrate the formatting and printouts of DFReport. Dummy data will be used.

List Records Unformatted

This function is by far the simplest way to get a hard copy of your datafile. Pressing L results in:

[PRINT OPTIONS MENU]

[A]LL RECORDS IN FILE
[S]ELECT INDIVIDUAL RECORD
[F]IND RECORDS WITH COMMON FIELD
[E]XIT TO MAIN MENU

POSITION PAPER IN PRINTER AT TOP OF PAGE

[PRESS THE APPROPRIATE KEY]

This menu functions exactly as the one in DFMail, with one exception. Instead of centering your mailing label, you are required to advance your printer to the top of the next page. Refer to the mail program for instructions on the above menu. A record printed unformatted will resemble the following:

[RECORD#1]-----
LAST NAME -----KONSHAK
FIRST NAME -----MIKE
CODE -----AUTHOR
STREET -----4821 HARVEST COURT
CITY -----COLORADO SPRINGS
STATE -----COLORADO
ZIP -----80917
PHONE -----303/596-4243

[RECORD#2]-----
LAST NAME - etc.

As you can see, the record data is printed in rows, which wastes considerable paper. Although this printout is quick-and-dirty, it can be cut out and pasted onto cards or filed in small cabinets or folders.

Pre-Defined Format

Pressing P results in:

LOAD FORMAT FROM WHAT FILE
? MAIL LIST

Enter the datafile format to be used for printing your report, then press the return key. The name of the last datafile loaded

in Datafile will be pre-printed for you after the prompt. Change the name by overstriking. The screen then displays:

DO YOU WISH TO REVIEW YOUR FORMAT AND/OR MAKE CORRECTIONS [Y] OR [N]?

Pressing Y sends you through the Define New Format routine. The current values of your format will be displayed. Alter by overstriking the values and pressing return. Also press return to accept the values. Pressing N gives you:

SAVE FORMAT [Y] OR [N]?

If you made any changes, go ahead and resave your new format by pressing Y. Keeping the same filename will scratch the old format. After N, you will progress to the Print Options menu, which has been previously described.

Define New Format

This routine creates a custom form based on your design. It would be a good idea to sketch out on a sheet of graph paper or programmer's pad what you want your report to look like. You will need to decide the following:

1. How many characters wide will the report be? Up to 136 characters may be printed, if your printer is capable of compressing text. Eighty characters is normal. Report widths less than 80 characters will be printed left-justified on the paper.
2. How should your title read? Up to four lines are possible, which will be centered at the top of the page.
3. How many columns will you

need? This will depend on which fields of your datafile you will want listed. Up to eight columns are allowed.

4. What is the width, in characters, of each column? This will depend on the combined character length of the record fields that you choose for each column. The total number of characters permissible in all the columns combined is 80 (or 136 with printers in compressed print mode), with two characters between columns. Choosing eight columns leaves you 76 characters for record fields (14 characters used in spacing).

5. Which record fields will be in each of the columns? As in formatting DFMail mailing labels, you will be able to combine up to three record fields in each column.

6. What will be the header name of each column? A header name cannot be longer than the chosen width of the column.

Try to remember the length of each field in the datafile that will be on this report. If the record data contained within the field is longer than the width of the report column, some end characters will be cut off.

Let's design a report using the datafile Mail List, which will give us a reference list of the records in the file. We will use first and last names (16 characters), street address (20), city (16), state and zip code (8), and phone number (12). This comprises a total of 72 characters, which we will put into 5 col-

umns (with 2 spaces between columns) for a total of 80 characters. The report will look like Table 1.

Now go back to the program to format the above report. Pressing D from the Report Printout menu sends you to:

[REPORT SIZE] UP TO 136
CHARACTERS WIDE

PRINTER MUST BE INITIALIZED FOR
WIDTHS GREATER THAN 80
CHARACTERS.
CHECK YOUR PRINTER MANUAL ON
HOW TO PRINT 136 CHRS

NUMBER OF CHARACTERS? 80

[TITLE FORMAT] PROVIDES FOR 4
LINES OF INFORMATION AT THE TOP
OF THE FORM:

TITLE #1? MAIL LIST RECORDS
TITLE #2? JANUARY 23, 1984
TITLE #3?
TITLE #4?

[COLUMN FORMAT] UP TO 8 COL-
UMNS WITH 2 SPACES BETWEEN
COLUMNS:

NUMBER OF COLUMNS? 5
POSITION OF COLUMN #1? 1
COLUMN #2? 19 <1 + 16 + 2>
COLUMN #3? 41 <19 + 20 + 2>
COLUMN #4? 59 <41 + 16 + 2>
COLUMN #5? 69 <59 + 8 + 2>

[HEADING FORMAT] COLUMN
HEADINGS CANNOT EXCEED WIDTH
OF COLUMNS:

COLUMN 1 HEADING? LAST/FIRST
NAME

COLUMN 2 HEADING? STREET
ADDRESS
COLUMN 3 HEADING? CITY
COLUMN 4 HEADING? ST & ZIP
COLUMN 5 HEADING? PHONE
NUMBER

CHOOSE WHICH FIELDS GO UNDER
THE COLUMNS

ENTER [0] IF ADDITIONAL FIELDS ARE
NOT DESIRED

1 LAST NAME	COLUMN 1	FIELD 1? 1
2 FIRST NAME		FIELD 2? 2
3 CODE		FIELD 3? 0
4 STREET	COLUMN 2	FIELD 1? 4
5 CITY		FIELD 2? 0
6 STATE		FIELD 3? 0
7 ZIP	COLUMN 3	FIELD 1? 5
8 PHONE		FIELD 2? 0
		FIELD 3? 0
	COLUMN 4	FIELD 1? 6
		FIELD 2? 7
		FIELD 3? 0
	COLUMN 5	FIELD 1? 8
		FIELD 2? 0
		FIELD 3? 0

DO YOU WISH TO REVIEW YOUR FOR-
MAT AND/OR MAKE CORRECTIONS [Y]
OR [N]? <N>

[SAVE FORMAT] [Y] OR [N] <Y>

SAVE UNDER WHAT FILE NAME?
? MAIL LIST

The program now jumps to
the Print Options menu for
choosing the records that are
to be printed. Now you should
refer back to the mailing label
program instructions.

MAIL LIST RECORDS

JANUARY 23, 1984

Last/First.Name	Street Address	City	St & Zip	Phone Number
Konshak Mike	4821 Harvest Court	Colorado Springs	CO 80917	303-596-4243
Mouse Mickey	1984 Disney Road	Orlando	FL 10001	800-555-1212
Bunny Bugs	21 Carrot Lane	Whatsupdoc	CA 99999	111-222-3333
Daniels Jack	555 Sobriety Blvd	Sourmash	TN 70707	000-876-5432

Appendix to Datafile

Programming User Programs

Datafile is flexible, in that you may write a subprogram that can be called from the Printer Main menu in Datafile. The basic ground rules are:

1. Subprograms cannot be larger than Datafile itself (approximately 7400 bytes).
2. Variable names used should not conflict with those that are necessary for maintaining the datafiles. Variable names used in counters, sorting routines and menus are safe to be duplicated. Try to mimic DFMail or DFReport in the way they handle data and perform operations. New variable names encountered may send the computer off garbage collecting.
3. Your subprogram should have the facility to load back Datafile so you can continue to update and manipulate your data.
4. Open printer and disk files properly when entering a routine. Ensure that you close the files before advancing to another routine or subprogram.
5. Include disk-checking routines to prevent program crashes. Check out any of the three Datafile programs for the routine.
6. It is easiest to modify or expand DFMail or DFReport instead of writing your own subprogram. You should safely be able to add 2000 bytes to DFMail and 1000 bytes to DFReport.

Variable Identification

The following is a list of all

the variables used in Datafile and its subprograms. Do not use these variable names except for accessing data. These variables never change in use or purpose.

R = number of possible records
X = number of current records in file

F = number of fields in each record

NF\$ = Name of current data or format file in memory

REC\$(R,F) = record data array

F\$(F) = field name array

L%(F) = length of field array

T%(F) = sorting buffer array

K%(R) = pointer array, keeps records in sorted order

ML\$(9,4) = array for combining fields in printing labels and reports

PC(10) = character position array for report columns

TT\$(5) = report title array

HC\$(9) = column heading array for reports

D\$ = chr\$(0) dummy string

CR\$ = chr\$(13) printer and disk carriage return

B1\$ = chr\$(10) printer line feed

B\$ = chr\$(32) 'space' character

E\$ = "EOF" end-of-file marker on sequential files

MEM = 31000 available memory (bytes) for record data

S, ST, EN, EM\$, ET, ES = disk error variables

The balance of the variables may be used in user subprograms, but should be avoided in additions to DFMail and DFReport. Counters and response variables are excepted. Check the programs carefully for conflicts.

I, J, L, N, M, Z = counters and

temporary buffers
 K = print routine pointer
 A\$, C\$, MR\$, DR\$ = responses from menus
 CK = check whether or not file has been saved
 RL = calculated length of record
 F1, F2, F3 = field pointer buffers
 HN\$, ID\$ = new disk header name and I.D.
 SB\$ = user subprogram name
 SF = field to be searched or sorted
 A1\$, A2\$, A3\$, A0 = buffers for loading disk directory
 PW = paper width of report (characters)
 CW = column width buffer
 RW = number of rows (lines) per label
 NL = number of lines for report title
 NC = number of columns in report
 PG = line counter for automatic paging of reports
 I\$ = input record selection
 T\$ = input common string to be searched
 B = tab for centering titles and first column of report
 LW = number of characters per row on labels
 T% = number of rows on labels

Printer Codes for Compressed Print

Many Commodore 64 owners have chosen to add standard ASCII parallel printers to their computer systems. These printers cost more, but have many capabilities and qualities that make the price secondary. Interfaces that convert the serial port on the 64 to parallel ASCII must also be purchased.

One feature that Datafile is able to utilize is that of compressed characters, allowing reports to be printed that have widths up to 136 characters. The Commodore 1525E and MPS801 do not have this feature, so you are limited to reports 80 characters (ten characters or columns per inch wide). Some printers with 15-inch carriages will print 132 characters in the normal mode, but will need to be compressed in order to print 136 characters on a standard 8½ by 11 piece of paper.

Table 2 shows the printer codes and procedures to use to set your printer into compressed mode. This should be done *before* you load and run Datafile (while you are in terminal, instead of program, mode).

If you are already into the program, and you want to send the printer commands, you must use the following procedures to keep from losing your datafile and pointers in memory:

1. While in the program, you must be at one of the many menus in Datafile, DFReport or DFMail. There should *not* be a flashing cursor.
2. Press the run/stop key. At the bottom of the screen, you'll see:

```
BREAK IN 30 (30 is the line number
                where the computer
READY          stopped the pro-
                gram)
```

[]

3. Type in your respective printer commands exactly as shown below if you are in the Datafile program. If you have

entered DFReport or DFMail, enter just the line that begins with PRINT#4. The printer files are already open when you are in these programs.

4. Type in GOTO 30, then press return. The number will be different, depending on which menu and subprogram you are in.

5. You will now be back in the program at exactly the place you left. To advance into the next part of the program, press one of the keys that the menu was previously showing. In some instances, you might lose part of your menu as the screen scrolls up, so try to remember which selection you want to press at this stage. E will normally exit you to the previous menu or send you to another program.

(NOTE: This technique may be used to send any printer

commands, not just compressed mode. Just make the appropriate changes to the CHR\$ codes.)

If you have a printer that is not shown in Table 2, review your manual for the proper printer codes. Other commands or modes that you might want to consider when printing reports or labels are Expanded (for making double size letters); Double-strike (for darker letters); and Changing Fonts (different letter styles).

(NOTE: Do not use modes that skip over the perforations in the paper. Labels do not need it, and reports are automatically paged by the program.)

Address all author correspondence to Mike Konshak, 4821 Harvest Court, Colorado Springs, CO 80917.

Type in the following commands exactly as shown to put your printer into compressed mode. Press return after each line.

GEMINI 10X:

96 characters (12 CPI):

OPEN4,4

PRINT#4,CHR\$(27)CHR\$(66)CHR\$(2)

CLOSE4

136 characters (17 CPI):

OPEN4,4

PRINT#4,CHR\$(27)CHR\$(66)
CHR\$(3)

CLOSE4

OKIDATA 82A:

132 characters (16.5 CPI):

OPEN4,4

PRINT#4,CHR\$(29)

CLOSE4

OKIDATA 92A:

96 characters (12 CPI):

OPEN4,4

PRINT#4,CHR\$(28)

CLOSE4

136 characters (17 CPI):

OPEN4,4

PRINT#4,CHR\$(29)

CLOSE4

EPSON RX80 F/T:

96 characters (12 CPI):

OPEN4,4

PRINT#4,CHR\$(27)CHR\$(77)

CLOSE4

137 characters (17.1 CPI):

OPEN4,4

PRINT#4,CHR\$(15)

CLOSE4

CITOH Prowriter:

96 characters (12 CPI):

OPEN4,4

PRINT#4,CHR\$(27)CHR\$(69)

CLOSE4

136 characters (17 CPI):

OPEN4,4

PRINT#4,CHR\$(27)CHR\$(81)

CLOSE4

NOTE: Changing CPI or pitch on daisywheel printers requires that a suitable daisywheel be installed. Although a command code may be sent, it is easiest to move the pitch switch on the keyboard to the proper setting (10, 12 or 15 CPI; i.e., 80, 96 or 120 characters on an 8½-inch paper width).

NOTE: Changing CPI or pitch on daisywheel printers requires that a suitable daisywheel be installed. Although a command code may be sent, it is easiest to move the pitch switch on the keyboard to the proper setting (10, 12 or 15 CPI; i.e., 80, 96 or 120 characters on an 8½-inch paper width). ®



Battleship War

BY KEITH MEADE

This battleship game challenges you on two fronts: Defend your fleet from enemy attack and learn this programmer's memory management tips to take advantage of the C-64's special graphics features.

You are in command of a battleship. Submarines silently move under the area you defend, seeking to attack your fleet's most vulnerable ships. Enemy planes pass overhead in a continuous stream.

You sit at the controls with clenched jaw and beaded brow. Blazing cannon fire and well-placed depth charges extract a heavy toll. For three tense minutes you batter the opposition hordes, then emerge in exultant victory, sporting a new high score.

Battleship War, an arcade-style game for the Commodore 64, begins with an instruction display that describes user controls and target point values.

**Run it
RIGHT**

Commodore 64

You must, as in any good shoot-'em-up, blast as many objects as possible—the smaller the targets the higher the point value. The submarines can be elusive, but if you watch them too closely, the airplanes will slip past you.

Playing for high score is definitely the way to go with Battleship War. The champion of this household is my wife; as of this writing, you'll need 2330 points to match her best score.

When typing the program, omit all remarks. Lines that end with 97-99 and contain asterisks may also be skipped. Notice that I substituted decimal points (periods) for zeroes, thus somewhat speeding up the program.

The game display for Battleship War is composed entirely of redefined characters. When you use custom character sets and the Commodore's other graphics features, memory management becomes a problem. Understanding where to store graphics data and getting the 64 to use it can be difficult. I'll discuss this problem and how I dealt with it in Battleship War.

I will conclude with a simple routine that lets you easily move screen RAM and open up a large area of free memory for sprites, character sets, high resolution displays and other features.

Defining the Problem

You should keep in mind that the C-64 features two separate systems sharing memory space—the video chip (VIC) and the Basic language. The VIC chip (not to be confused with the VIC computer) handles all data and operations that relate in any way to the video display. The VIC, however, has a limitation for which you must allow—the chip can access only 16K of memory. I will refer to this memory as the VIC video bank.

Within this bank all data related to the video display must reside. Within the VIC video bank is an area I call screen memory, 1K of RAM that contains data for the standard text display. You will often see screen memory referred to as the video matrix or character pointer memory.

(Be sure at this point that you understand the concept of the 16K VIC video bank. Do not confuse it with the screen memory, which is only a portion of the video bank.)

With 64K of total memory, there are potentially four VIC banks. Normally, the VIC chip is accessing the first bank. It sees the memory from addresses 0 to 16384. A look at a memory map reveals that this is a busy area. The only large chunk of free memory is within the space used by Basic, but unless you really know what's going on, only use Basic memory for a Basic program.

So, the major memory management problem is this: There

is not sufficient available space to allow use of the 64's special graphics features. In the first VIC video bank, the bulk of unused memory is reserved by the Basic system for Basic programs. As it turns out, both Basic and the VIC chip are willing to compromise. The Basic space can be trimmed on either end. The VIC chip can look at any of the four 16K video banks.

The simplest solution would seem to be moving back the beginning of Basic's program area. Basic programs normally start at address 2048. The following sequence would free up 4K bytes.

POKE 6144,0:POKE 44,24:NEW

Basic requires that the first byte in its program area be zero, so zero is the first Poke. The second Poke sets back the beginning-of-Basic pointer. The New command causes the program area to be straightened out within its new boundaries.

I've seen this method successfully used many times, but there's an obvious drawback. A program can't reserve memory for itself; there must be a separate set-up program or you must manually type in the configuration sequence. As long as there are alternatives, this technique should be unacceptable.

The beginning of the Basic program area actually contains the first lines of the Basic program itself. It's certainly understandable that we can't cut off that chunk of memory without destroying our program. So

what about the other end of the Basic area?

The very top of Basic memory is used to store the values of variables (specifically, string variables). Stealing from here is going to zap the variables, but notice that the program itself will survive.

POKE 55,0:POKE 56,128:CLR

The Poke commands set the top-of-Basic pointer down 8K bytes. The CLR command forces Basic to rebuild its variable system at the new, lower location. After execution of this command, all string variables are null and all numerics are equal to zero.

Clearly, the drawback to this method is manageable and, as you will see, I recommend going with it. The secret is to reserve the memory before you declare or use any variables. To be safe, devote the first line of your programs to the function of clearing this space, if needed.

The Basic command sequence in the last paragraph sets the top limit of Basic at address 32768. Remember the VIC video banks? Well, the third bank begins at address 32768. Perfect!

POKE 56576,5

Believe it or not, this instructs the VIC chip to take all data from the third video bank. So, while you've lost 8K of Basic RAM (I've never written a program anywhere near 30K bytes long), you've gained free and clear memory for use of custom graphics.

You Can Bank on It

Unfortunately, the actual structure of the new video bank is complicated, but bear with me. You don't need to understand it at all if you're willing to abide by the rules and address boundaries I'm presenting.

From address 32768 to 36863 is a 4K chunk that can be used normally in any way you see fit.

Addresses 36864 to 40959 are the VIC chip's 4K "blind spot" in this video bank. The VIC ignores the RAM in this range and, instead, sees the character set ROM, which contains the definitions for the two standard Commodore character sets. Peeks and Pokes in the Basic program will see the RAM. You could use this to store data or machine language in the RAM.

Addresses 40960 to 49151 look like 8K of RAM to the VIC video chip. A Basic program Poke to this area stores values in that RAM, but a Peek sees data in the Basic system ROM chip! It's confusing, but this is actually a very large and useful stretch of memory.

Just remember that you can Poke data in, but you can never Peek that data. (It's probably not an appropriate region for screen memory, but when would you ever need to Peek at a redefined character set?) Sprites, too, would oftentimes be fine here. Machine language programmers would be able to examine this memory by switching out the Basic ROM, but that's not possible in a Basic routine.

Screen memory, as you re-

call, is the 1K containing the text screen data. You may define screen memory as being any one of the 16-1K areas within the video bank. A character set is 2K in length, and any of the 8-2K regions of the video bank may be designated as containing the character set.

In Battleship War, I used variables to hold these location values. SCRAM could have a value of 0-15. SCRAM = 0 means that screen memory resides in the first 1K of the VIC video bank (thus, beginning at 32768). CHSET (value equals 0-14, even) specifies which 2K area holds the character set. Odd values of CHSET simply have the same effect as the next even number below.

The VIC chip contains a single register that sets the locations of screen memory and the character set. The following command will work for all meaningful values of SCRAM and CHSET.

```
POKE 53272,16*SCRAM + CHSET
```

The Basic system must be separately informed of the location of screen memory so that it can properly handle screen input and output. The following command will perform this.

```
POKE 648,128 + 4*SCRAM
```

Graphics Routines

Investigation of other graphics features should reveal how they fit into this memory configuration. Observe in the program Battleship War how the system has been specifically implemented.

Below are the routines from Battleship War that could be used in any program to relocate the video bank and open up space for your special graphics data.

```
10 POKE 55,0:POKE 56,128:CLR  
(Remember, it is best to have this be the first program line.)
```

```
10000 IF SCRAM<0 OR SCRAM>15  
      THEN PRINT"SCREEN RAM  
      ERROR":STOP  
10010 IF CHSET<0 OR CHSET>15  
      THEN CHSET = 4  
  
10020 POKE 56576,5  
10030 POKE 53272,16*SCRAM + CHSET  
10040 POKE 648,128 + 4*SCRAM  
10050 RETURN  
  
10100 POKE 56576,7  
10110 POKE 53272,20  
10120 POKE 648,4  
10130 RETURN
```

Save these program lines and use them. They'll make your life a lot easier, believe me.

To rearrange the video, set SCRAM equal to 0-15, CHSET equal to 0-14 (even) and GOSUB 10000. The new location of screen memory will be $32768 + 1024 * \text{SCRAM}$. The VIC chip will expect to see the character set at $32768 + 1024 * \text{CHSET}$. CHSET = 4 will point the VIC to the standard character set (6 for upper/lower-case).

Sprites or other graphics features may easily be used by keeping the previously described VIC memory bank structure in mind. Remember, in particular, that the sprite image pointers are part of the screen memory and move with it.

To restore the usual configuration, enter GOSUB 10100. The

default screen memory area at location 1024 is not disturbed by any of this activity. Toggling between the two subroutines offers a simple method of page flipping, with which you might wish to experiment. I'll conclude this discussion with a little demo program to get you thinking. Type in these lines along with the above routines (don't forget line 10).

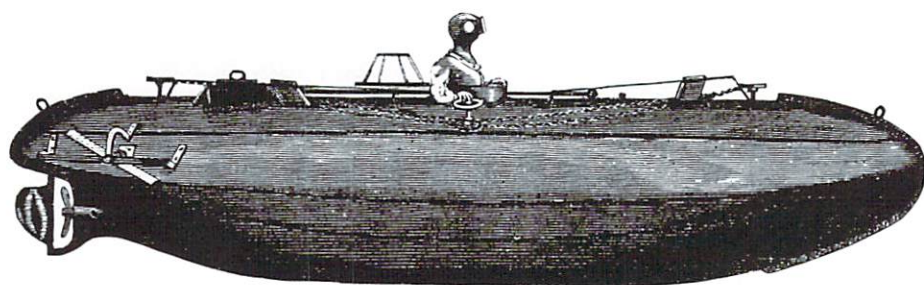
Before you run the program, notice that the Print statements are only executed once.

```
100 PRINT "(clear screen)"
110 PRINT "THIS IS THE OLD
    SCREEN."
```

```
120 SCRAM = 0:CHSET = 4:GOSUB
    10000
130 PRINT "(clear screen)"
140 PRINT "HERE IS THE NEW
    SCREEN!"
150 FOR D = 1 TO 1000:NEXT
160 GOSUB 10100
170 FOR D = 1 TO 1000:NEXT
180 GOSUB 10000
190 GET AS:IF AS = " " GOTO 150
200 GOSUB 10100:END
```

Press any key to end the program. I hope you can put these routines to good use. Don't forget to try Battleship War. 

Address all author correspondence to Keith Meade, 3111 15th Ave. NW, Rochester, MN 55901.





Slide

BY ROBERT ROSSA

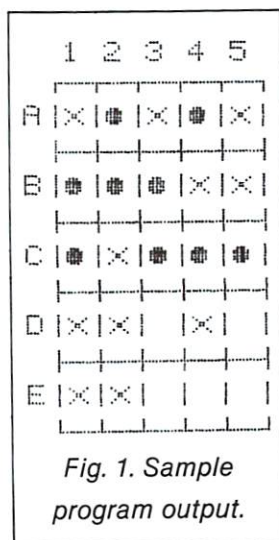
Five in a row tic-tac-toe may sound easy, but don't be deceived. This game offers more challenge than perhaps you're willing to tackle.

Slide is a simple strategic board game that pits you against the computer, which has been programmed with a greedy strategy and rarely loses.

Slide is played on a five-by-five square board with 25 cells. You and the computer take turns entering single tokens from the top (numbered 1-5) or the left (lettered A-E). A token entered into a row or column will slide tokens already present on the board over or down one cell. Tokens may be shoved off the game board. The first player to have five tokens in a row, either horizontally, vertically or diagonally, wins.

On the game boards drawn by this program, the computer has the crosses and you have

the circles. In the sample position shown in Fig. 1, you win (horizontally) by making move 2, but lose (diagonally) by making move C.



Making the Best Move

The first few moves of the computer are random, to provide some variety in the games. You can select how many moves in advance the computer can consider—either two, four, six or eight moves ahead. Obviously, the more moves ahead, the more time it will take the computer to make a selection. You can gradually advance the level of play as you learn the game.

To determine its best move, the computer must look at the game tree (see Fig. 2). Each

**Run it
RIGHT**

Commodore 64

move is drawn as a branch of a tree. Note that the tree is drawn with its branches hanging downward. Each path downward from the root (at the top) represents a possible sequence of moves. Each move has a value, so you can pick the move with the largest value.

For speed, the move-selection logic is written in machine language. I wrote the algorithm in Basic and then translated it into machine language. In Basic, at level 4, I could mow the lawn while the computer was deciding its next move. Another factor allowing more speed is the use of pruning; we don't have to consider all possible moves four plays ahead.

Each position is given a numerical weight, which is intended to measure a player's advantage. If the player who just moved has a win, the weight is as large as possible, in this case, 32.

If the opposing player wins, the weight is -32 , as small as possible. Otherwise, the weight is the advantage the player who just moved has over his opponent.

There are ten possible moves from any position. How can a player determine the best move? Suppose you are looking just two levels ahead. For each of your ten possible moves, you look at what your opponent can do. For each of your possible moves, your opponent has ten possibilities, so there are 100 combinations of moves.

For each move you can make, you want to know how well your opponent can do. So you assume that your opponent will select a move by picking the position of maximum weight. The value of your move is then the negative of this maximum weight.

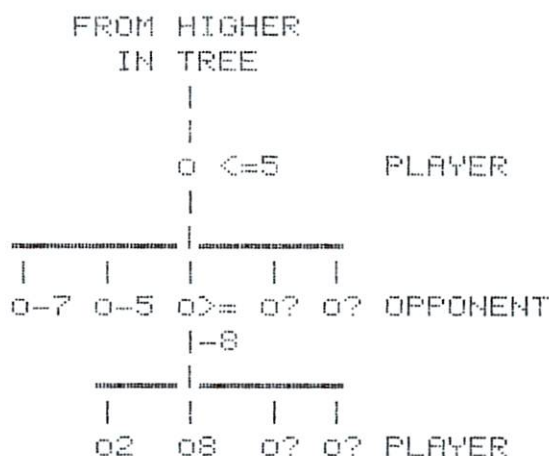


Fig. 2. Game tree, which represents a possible sequence of moves.

Pruning the Game Tree

If you search four levels ahead, it seems that you might have to look at all 10,000 sequences: you move, then your opponent moves, then you move again, then your opponent moves. For each of the 1000 possibilities at level 3, the value is the negative of your opponent's best possible move.

For each of the 100 positions at level 2, the value (to your opponent) is the negative of *your* best possible move. Finally, at level 1, the value of each of your ten possible moves is the negative of your opponent's best possible moves.

If you search six levels ahead, you'll have 1,000,000 sequences at which to look. Clearly, this is a tedious task even for a computer. Fortunately, there's a way of eliminating most of the possible sequences—it's called pruning the game tree.

For example, consider Fig. 2. Suppose it represents your knowledge about a particular part of a game tree just as you finish finding the value of the second game position in the bottom row. When you find that this value is 8, then you can say that the value of the position just above it (its parent) can't be better than -8 for your opponent.

Since your opponent already knows he has better moves, including one with a value of -5 , there is no need to continue evaluating the other possibilities (children) of the position. At this point, you can prune the

game tree, that is, stop considering the -8 position and move on to the position to its right in the figure.

You'll be able to follow this pruning process when you play the game, since the program places the moves it's considering on the screen.

The Program

To load and play the game, you need only enter the Basic listings. The first Basic program (Listing 1), consisting mostly of Data statements, Pokes the machine language needed by the game. The second program (Listing 2) plays the game.

Listing 2 begins by protecting the machine code. Then it gets the game tree's depth of search by asking you to choose a level.

The variable CC, initialized in line 90, counts the moves made by the computer while it is playing randomly. The control for random moving is in line 280. The variable BQ is the move number, which is needed by the subroutine that updates the screen and internal game boards.

Lines 100–120 decide which player starts. AQ is a code for the current player; 1 is you and -1 is the computer. Line 130 is executed if the computer moves first; it sets up the game board and makes the computer's first move. Line 140 sets up the game board if you move first.

Lines 150–240 obtain a player's move, recode it for the routine that updates the board and then call that routine.

The game board is coded in memory in the 25 bytes starting at location 28672 (A in line 470). When the search routine is called, it calls itself recursively. It needs up to eight copies of the game board, depending on the level you choose. These are located in the 200 bytes beginning at location A.

The board evaluation routine, called from the Basic program by SYS RT in lines 260 and 320, needs to know which of these eight boards it is supposed to evaluate. This is the purpose of location LZ, which is Poked in lines 260 and 320.

The evaluation routine returns a 1 in location CZ, if you have five in a row; otherwise, it returns a 0. Location CZ + 1 is the return for the computer, so lines 260 and 270 check to see if there is a winner. S4 and T4 keep track of the number of wins for each player.

Line 290 calls the tree search routine that selects the computer's move. BQ is the move picked; if the computer has only losing moves, you force move 1 in line 300. Line 310 then updates the board. Lines 320-340 check for a winner and then cycle back for the next move. Lines 350-360 contain the logic for a random move by the computer.

The subroutine at 370-520 sets a number of constants: S, the starting location of the screen; DN, the screen width; MR and MC, the starting row and column for the game board on the screen; LC, the starting screen RAM location for the

board; D2, twice the screen width; and the characters used to draw the board.

Certain vital memory locations are given symbolic names in line 470. In line 510, the internal board at level 0 in array A is initialized. Line 510 is also executed before each new game.

The subroutine in lines 530-710 builds the screen, using the symbolic parameters set up by the initialization routine. The routine in lines 720-1070 updates the board after a move. The level 0 board in array A is looked at by this routine. Peeks and Pokes must be used, since A is not a Basic dimensioned array.

Finally, the last few lines at 1080-1130 print the score and provide for a continuation of the match. I have found the match at level 6 to be pretty uneven—in the computer's favor. 

Address all author correspondence to Robert Rossa, 1901 Starling, Jonesboro, AR 72401



Mystery of Lane Manor

BY JIM SANDERS

Where are Sherlock Holmes, Nero Wolfe and Peter Wimsey when you need them? Someone's been killed at Lane Manor, and you're going to need more than a slick trenchcoat and a funny hat to discover *where, with what and whodunit.*

Mystery of Lane Manor is a whodunit game, where the players act as detectives trying to solve the mysterious murder of industrialist James Lane. The mystery is solved (and the winner declared) when the murderer and weapon are discovered and the location of the crime is determined. The correct answers are randomly generated each time the program is run, so the game provides an endless source of mystery.

**Run it
RIGHT**

Commodore 64
2 Joysticks

There are six different rooms where the crime could have occurred. There are five different people who could have murdered Mr. Lane. And there are five weapons that could have been used.

How to Sleuth

To make a guess, each player, in turn, moves a token to the red square in a room.

The step-generator, which is running when the play screen is initially displayed, determines the number of steps you move your token. The player whose name is displayed goes first, by pressing the fire button. This stops the step-generator, and an arrow reveals the number of steps you must move.

Once you've reached the red square, the program will enter the Guess routine, and the list of suspects will be displayed under your name. Move the flashing arrow (via the joystick) to the number corresponding to the suspect you deem guilty, and push the joystick's fire button to register your guess. Next, the six possible rooms will be listed; guess again and press the fire button. The list of weapons will then be displayed.

After you've made your three guesses, a review of these guesses will be displayed and the number of correct guesses will be revealed.

For an added challenge, at the beginning of the game you're given the option of seeing or not seeing the step-generator pointer. With the pointer invisible, planned movement through the manor is practically impossible. After you make a guess, the token is placed somewhere in the main hallway.

Look Out!

The manor is not without its own hazards. Trapdoors randomly spring open and can become very troublesome. If you fall through a trapdoor, you are forced to begin the trek again, from the home position. As the game progresses, the trapdoors may block doors or eliminate needed guessing squares. You may clear the

manor of the trapdoors by pressing the return key. If you do this, both tokens are forced to begin again at the home position.

You may discover all the data to solve the mystery, but your final guess must be made in the room where the crime was committed.

When the mystery is finally solved, the winning detective is congratulated with a musical fanfare and the time it took to solve the mystery.

For your convenience, I've included an itemized list of instructions, which should make learning the game easier. I hope you enjoy playing Mystery of Lane Manor.®

Address all author correspondence to Jim Sanders, 12629 S.R. 347, Marysville, OH 43040.

1. One or two detectives may work on the case.
2. The object is to solve the murder in the shortest amount of time or, if two players, before your opponent.
3. Murderer, room and weapon must be found.
4. In order to make a guess, you must be in a room and on the red square.
5. You must move your token the number of steps given by the step-generator.
6. After you've made your guesses, you'll be informed how many are correct.
7. After a guess, your token will be placed in the safety of the main hallway.
8. To win, you must solve the murder in the room where it occurred.
9. Guesses are made using the guess-selector, via the joystick and the fire button.
10. The fire button is also used to stop the step-generator.



Money Grubber

Formerly "Taxman."

BY DOUG SMOAK

You've got to be quick to stay ahead of that money-grubbing taxman, who's hot on your trail. He's after your every dollar and won't stop at that. He wants your life.

In Taxman, you must move through different levels of the screen and gather money. But just as in real life, someone else wants your money, too. The taxman, of course. And just as in real life, the taxman wants more than your money. . . he wants you!

To play Taxman, you must plug a joystick into the rear port of your C-64. You earn points for each \$ that you gather and you lose points for each one that the taxman gets. You begin with three lives (maybe nine would be better?) and each time the taxman catches you, you lose one life. If you reach 2000 points, you gain a life. The program makes a noise when you get caught and a beep when you gain an extra life.

**Run it
RIGHT**

Commodore 64
joystick

The Game's Ingredients

One of this game's interesting features is the music, which plays continuously throughout the game. It is in machine language and driven by interrupt, so it doesn't slow down the game.

Another interesting feature is the large alphabet, which is used in the title and score displays. It is made of strings that print graphics characters to build each letter. The program doesn't use every letter, but all the letters are included in lines 1110-1510.

The routine at lines 1530-1540 performs the actual conversion of G\$ to large letters. By using this routine and the string array of the alphabet and numbers, you can print large text in your own programs.

To give the illusion of a man running, I use four custom characters, two for each direction (left or right). To determine which character to display I use an interesting technique. In line 310, the character is chosen by the expression $CH = PEEK(C) + (MEAND3)$. The value of $PEEK(C)$ is set by the machine language routine that reads the joystick and determines whether or not you have moved. The $(MEAND3)$ is what makes the character change.

As ME changes, ME AND 3 go through the sequence 0,1,2,3 or 3,2,1,0, so the character that is displayed is changed as ME is changed. Since ME is the runner's position on the screen, the runner is animated as he runs. A similar method is used in line 300, to animate the taxman through his two positions.

You must actually enter two programs to get Taxman off and running. The first one, Tax Loader, sets up the custom character set that will be used in the game, and then loads and runs the game program.

Since you can use only 12 custom characters, I wrote a short machine language routine that first moves the entire ROM character set into the RAM area, which the game will use. I then used Basic Pokes in the data, for the 12 characters to be redefined. The first five data lines of the Tax Loader hold the machine language routine, and if you don't enter it correctly, line 20 will end the program and tell you the checksum is in error.

You don't want an error in these lines, since the machine language routine disturbs interrupts and changes location 1 to "bank out," or switch out, I/O memory, and switch in the ROM, so it can be read. Be sure, therefore, to get these five lines correct. If the rest of the data is not correct, the characters will not be correct, but the program will not bomb.

Also note that line 70 must be set for either tape or disk, and if you are using a tape ma-

chine, you *must* have the Taxman game program as the next program on tape after the Tax Loader. You might even want to put two copies of the program after the loader, in case the recorder misses the first one.

You should also be careful with the data in the game program, since most of it is for machine language routines and could cause the 64 to crash if you don't enter it properly. I won't try to explain the program in detail, but will give you a brief description of what each section does.



Line by Line

Line 0 correctly sets the end-of-program pointers after you've loaded the loader program and run the game program.

Lines 10-180 set up variables, print the title and read and Poke in the data for the machine language routines that are used.

Lines 190-360 make up the play loop.

Lines 370-480 set up the strings that put the money and the "holes" on the screen.

Lines 490-580 print the play screen for each level of play.

Lines 590-630 animate the characters on the screen.

Lines 640-790 update the score, check for end of game, check for "bonus life" and display the score at the end of the game.

Lines 800-810 determine the skill level at which you start.

Lines 820-830 initialize the SID chip.

Lines 840-1100 are the data for the machine language routines.

Lines 1110-1520 are the strings that contain the large alphabet.

Lines 1530-1540 print the large letters from G\$.

Lines 1560-1580 play the opening fanfare. [R]

Address all author correspondence to Doug Smoak, 303 Keyward St., Columbia, SC 29201.





Touchdown

Formerly "NFL Football."

BY LARRY D. SMITH

It's first and ten, do it again! Surprise! You've been drafted into the NFL as a starting quarterback. How well can you rack up those points and smother your computer opponent?

If you're tired of shooting at or being chased by aliens, you might like to try quarterbacking an NFL football team.

NFL Football, a game that runs on the C-64 or on the VIC-20 with any memory expansion, simulates the situations that commonly occur in professional football games. While the graphics usage is not astounding, the computer is a competent opponent and will maintain your interest as you lead your team to either victory or defeat.

The use of the Kernal plot routine for positioning the cursor, and of the random number

techniques for simulating the statistics of a football game, are programming tips you may want to extract. The computer is well informed about the game of football, and the game situations are so realistic that you'll probably find yourself following the strategies of the pros.

Butting Heads

Table 1 lists the offensive and defensive plays available to you and the computer. As the table illustrates, the offensive player has six plays to choose from: two running plays, two passing plays and two kicking plays. The defensive player may choose to defend any of the running or passing plays.

From the draw play to the long pass, the plays increase in yardage potential. However, with the potential comes risk, as the possible yardage loss also increases. For example, the draw play typically makes a three- to five-yard gain, or a one- or two-yard loss. The long pass, on the other hand, may make as much as 70 yards, but could result in a ten- to 15-yard loss. Also, the computer generates occasional turnovers, with increasing turnover frequency for the plays with the greater yardage potential.

The computer determines the kicking game purely from random numbers, though it uses a non-uniform distribution to

**Run it
RIGHT**

Commodore 64 or VIC-20
with 3K, 8K, or 16K expansion

typify a professional game. The running and passing play yardage is determined both by random numbers and the match or mismatch between the chosen offense and defense.

If the chosen defense matches the offense, the yardage gains are substantially reduced. If the offense and defense are considerably mismatched (like defending for a long pass when the opponent runs a draw), then the yardage gain is generally increased. This provides a good mixture of chance and strategy.

Get a Kick Out of It

The computer plays the field goal kicking game by professional rules. If a field goal attempt is missed outside the 20-yard line, the opponent receives the ball at the line of scrimmage. If the field goal is missed inside the 20-yard line, the opponent gets the ball on the 20.

The computer also knows about safeties, so be careful when running the ball from deep in your own territory. Punts that go into the end zone are placed on the 20-yard line.

While the computer randomly chooses its offensive and defensive plays, it is aware of the time remaining, the score and the field position. On fourth down, the computer generally will punt (or if close enough, try for a field goal), but if it's behind late in the game, it may go for the first down. The computer also knows that a trailing human opponent is apt to

throw a few long bombs, so when it's ahead, it tends to defend against the longer potential plays.

The screen display consists of a 20-yard section of field around the scrimmage line, the first down markers and a scrimmage-line marker. It also displays the time (number of plays) remaining and the number of downs.

When you're prompted for a play, you enter the numeric code (from Table 1) of the desired play, or a question mark if you've forgotten the plays. A question mark will result in the computer displaying a list of options.

CODE	PLAY
1	Draw play
2	End sweep
3	Short pass
4	Long pass
5	Punt (offensive play only)
6	Field goal (offensive play only)

Table 1. The offensive and defensive plays available to you and the computer, and their numeric codes.

Choose Your Foe

You can customize NFL Football to play against your favorite opponent. Just replace all occurrences of "ME" and "I" in the program listing with the name of your chosen opponent.

Try to retain the same spacing before and after the name so that the display will look well. Caution: Do not change the numeric variable ME, which is *not* inside quotation marks and, if changed, will cause untold problems.

C-64 occurrences of "ME" and "I" are in lines 100, 195 and 230. VIC-20 occurrences of

"ME" and "I" are in lines 100, 190 and 230.

After you insert your opponent's name, just run the program and enjoy the thrill of battling to win the Big Game.®

Address all author correspondence to Larry D. Smith, 5404 Inspiration Lane, Las Cruces, NM 88001.



Speller

BY GARY FIELDS

In this program, your child's ability to learn words and their definitions isn't measured by points. Noises, clues and a smoke-puffing, chugging sprite train alleviate the pressure and make learning fun.

I began writing the 64 Speller to help my seven-year-old daughter with her weekly spelling lessons. I wanted the C-64 to prompt her to spell a word, then to check to see if she had spelled it correctly.

The major problem was coming up with an interesting and usable prompt, one that wouldn't display the word. A speech synthesizer would have been nice, but I didn't have one. The solution was to offer a definition of a word, then let her spell the word it defined. This approach turned up a plus, because the display of the definition increased the learning.

Run it
RIGHT

Commodore 64

What's It All About?

This program not only reinforces spelling, but also knowledge of word meanings and awareness of the keyboard. And the 64 Speller is enjoyable. It's full of sound—nice sounds when the child's spelling is correct, not-so-nice sounds when it's wrong.

This program is friendly, too. It first displays all the words that will be in the program and lets the child study these for as long as he or she pleases. Once the child gets into the actual program, it remains friendly with aids.

The word definitions are slowly scrolled across the screen with attention-keeping clicks, and other prompts are announced with a tone.

Pressing the F1 key provides the child with clues. It gives aid one letter at a time, repeating clues after each spelling try, and also adding letters after each try, up to and including the total word.

And, of course, there's a reward for getting the word right—Casey Jones rolls along "on the right track" in his smoke-puffing and sound-chugging locomotive. The program reinforces the correct spelling with a final toot of the train whistle.

This continues until the child correctly spells all the words in

the program's memory. The program then says goodbye with a hearty "Well done!" followed by a final review of the spelling words.

Providing the Words

The words are placed in the program by a parent or teacher. The program prompts the correct entry through a special "change WORDS/DEFINITIONS F2" routine (lines 80 and 155).

Words and definitions should be entered in even numbers because the data display is read in pairs. So, if you enter 11 words and definitions, add another to make an even 12. Also, the definitions should be less than 40 letters in length (try a longer one and you'll see why).

Our practice at home is to duplicate each week's spelling-lesson words into separate programs titled Speller 1, Speller 2, and so on. One disk is reserved just for spelling words. That way, the child can go back and try old lessons again. Or, if you're like us and have a younger child, the saved lessons may be for his or her future.

Descriptions of Lines

Line 15. Sets the screen color and switches to upper/lower-case.

Line 20. Sets the basic sound and sprite-generating variables.

Line 25. Puts the data-reading pointer to the 0 in line 2950.

Lines 50-95. Title page.

Lines 80 and 155. Prompt for the word-replace routine.

Line 87. Reads and Pokes the

train into memory.

Lines 100-165. Display the words this program will review.

Line 190. Makes the data pointer look at the first words.

Line 210. Reads the first word and definition; cc is the clue-variable counter.

Line 215. Looks to see if all the words have been used.

Lines 225-270. Slowly scroll the definition (b\$) with clicks.

Line 280. Asks for the word defined.

Lines 300-310. Check the spelling and go to the correct or incorrect routine.

Lines 600-790. The correct spelling routine.

Line 610. Turns on the sprite and expands it.

Line 612. Makes sprite black.

Lines 622-624. Draw the tracks.

Line 625. Makes the train move from right to left.

Line 644. Chugging sound.

Line 645. Places the train in the correct starting location.

Line 646. The smoke variable is sm.

Line 651. Turns the chugging sound off.

Lines 657-661. The two train toots.

Line 662. Turns sprite off.

Lines 730-790. Make smoke come out of train's stack.

Lines 800-910. Incorrect spelling and clue routine.

Lines 810-840. Buzzer.

Lines 892-894. Check for clue request.

Line 895. Prints clue using LEFT\$ command.

Line 910. Checks to see if new spelling is correct.

Lines 1500-1820. The win routine and sound.

Lines 1830-1890. Display the word list for the last time.

Lines 2000-2030. Beep tone sound.

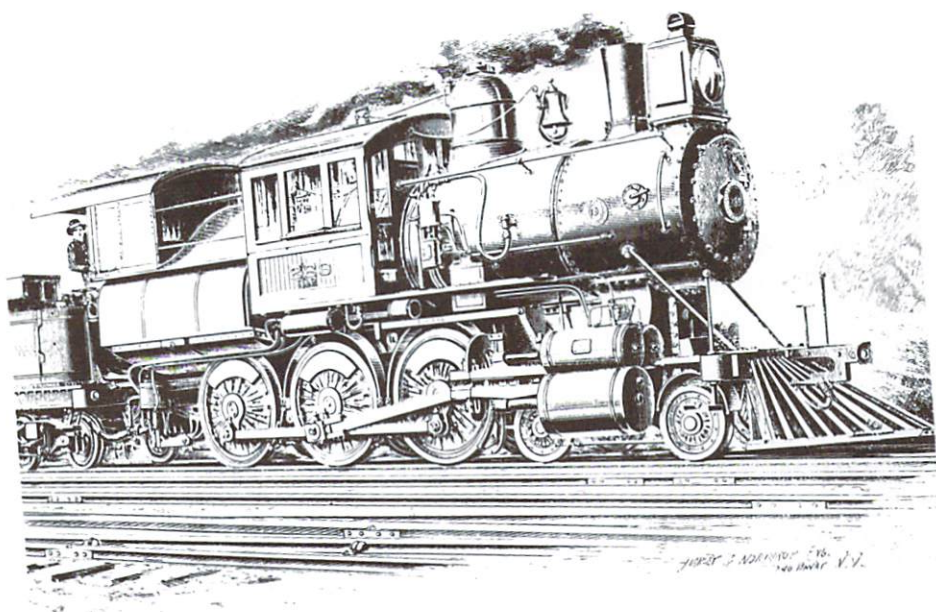
Lines 2500-2900. Aid routine to replace data.

Lines 2950-2953. Train sprite data lines.

Lines 3000-4999. Replaceable data lines.

Line 5000. End of data line—never replace this line. 

Address all author correspondence to Gary V. Fields, 86 Lanvale Ave., Asheville, NC 28806.





The Many-Colored VIC

BY TOMMY MICHAEL TILLMAN

This editor is an artist when it comes to designing and modifying your multicolored VIC-20 graphics characters. What's more, it's easy to use.

I've seen many fine articles on multicolor programming, but I've never found an easy-to-use Editor that would design these types of characters. As a result of my futile search, I wrote the Editor program.

This program is designed to work on a VIC with 3K memory expansion (Super Expander will also work). Simply type in this program, save it and use it.

If, however, you have an 8K or greater memory expander, type in both the Screen Relocation program and the Editor program and save each separately. Whenever you wish to use the Editor, load the Screen Relocation program and run it. This will make the VIC look like it did before you added the 8K

memory. Next, load the Editor and remove line 10 (in fact, you may remove it permanently and save the new version). Now you may run the program.

In the upper left-hand corner of the main screen is displayed the command board. If you don't know what to do, wait for a command to pop up! The following is a list and description of the commands.

C—Change colors

D—Display

G—Go to a new character number

L—Load an old character set

S—Save the character set (printer-screen-tape)

Remember that these are the main commands. If you choose one of them, they'll all be erased from the screen and replaced with new subcommands, which will give you instructions on how to continue properly. Always wait for the commands to appear! (There is a slight time lag in some subroutines.)

If you look to the right-hand corner of the screen, you'll see a large 6 × 6 square. This is the display area, which allows you to put your newly constructed multicolor characters on top of or beside each other to create larger multicolor characters.

If you look at the middle of the display screen, you'll see three rows of normal high-resolution characters. These

**Run it
RIGHT**

VIC-20 with 3K expansion
Datassette
Printer (optional)

are the characters that you may restructure into multicolor characters. (As you create a new character, the character corresponding to the one you're working on will change shape. The characters in these three rows will still be displayed in High-Resolution mode.)

In the left-bottom corner of the screen, you'll see the four colors with which you may color your multicolor character: screen color, border color, character color and auxiliary color. (These colors will be known, respectively, as color 1, color 2, color 3 and color 4.)

In the right-bottom corner of the screen, you'll see the character you're restructuring. It will be displayed in High-Resolution mode and, below, in its appropriate Multicolor mode.

The bottom middle of the screen is the most important. This is the work area where you'll display the multicolor character in a 4×8 display. The character will be made up of 32 large blocks, colored in one of the four colors you're allowed to use.

Around the top and left of the large character are arrows that indicate which block of color in the work area you'll be changing. Press the cursor keys to move the arrows. The right cursor moves the arrows right and the left cursor moves them left. Likewise, the down cursor moves the arrows down and the up cursor moves them up. The character number on which you're currently working will be displayed over the work area.

How To Use the Program

Load the program according to previous instructions. Run it and wait for the screen to set itself.

Now select a character to work on (0-57). Press G for Go to Character and then input the character number (0-57). Always press the return key after responding to requests for input. Also, for later reference, don't forget to make a note of what characters you are changing.

If you wish, you may change colors now. (In fact, you can change the colors anytime you are back to the main command screen.) To change colors, press C. Input your choice of screen color from the color list (see Table 1). Remember to press the return key after you input your choice.

Next, select your choice of border color, then character color, and finally, auxiliary color. (You may choose only colors 0-7 for character color.)

Now you may restructure your character. To do this, you must use the cursor keys and the number keys 1, 2, 3 and 4. Notice the arrows above and to the left of the character. These arrows indicate which color block of the character you are changing. By pressing the cursor keys (with the shift key) you may position the arrows to point to any block within the work area.

To change the color of the block, you must use keys 1, 2, 3 or 4. If you press key 1, you'll erase that block (because you

are coloring it in the background color; key 2 will color that block in the border color; key 3 will color it in the character color; key 4 will color it in the auxiliary color.

Notice that as you change the blocks, the corresponding pixel dots in the multicolor example change to the proper color! The corresponding dots in the high-resolution example change to the proper configuration, too.

If you wish to display your multicolor characters on the display screen (upper right-hand corner), then press D for the display function. First you will be asked for the width and height of the display screen (the number of characters horizontally and vertically). Then you'll be asked for the character number and the character color for that particular character. Repeat this information until the display screen is full. Then you will return back to the main commands.

Note that if you change the four main colors by using the color command, then the screen, border and auxiliary colors for all blocks in the display will also change. Each block's character color will stay the same, though, because the character color of each block is independent of other colors.

To save your data for each character, press S. The screen will clear and you'll be presented with three options. If you choose T for tape, insert in the Datassette the tape to which you wish to save the

character set and press T. Next, input a filename and press the record and play buttons on the Datassette. Stand by until the character set has been copied to tape. There will be a slight delay until the main screen is once again displayed.

If you choose P for printer, then stand by while the character set is copied to the paper. The output will be as follows. The first number in each line is the character's number. The next eight numbers are the byte numbers that represent that character in the character set. (The first eight numbers in the set are for drawing character 0. The next eight numbers are for character 1. This continues all the way to character 57.)

Now you should make a mark beside the characters that you've changed. You should also make a note of the colors you're using and the character color you are using for each character.

If you choose S for screen, then the output will be identical to the printer output, except that only seven characters at a time will be displayed. You may copy onto paper the pertinent information that you desire. You'll be returned to the main screen after you finish going through all 58 characters.

If you wish to reload a character set for reviewing or modification, then press L. Insert the appropriate tape into the Datassette, type the name of the character set and press the return key. Press the play button on the Datassette and

stand by while the character set loads. When the character set is ready, you'll be returned to the main command screen.

To quit, press Q. On a VIC with 3K memory, everything will be fine (including the new character set, which will be in memory locations 7168 to 7679).

On a VIC with 8K or more memory, however, don't use Q unless you permanently modify line 370. Simply delete everything between the words THEN and END. Now the VIC will work normally.

How Multicolored Characters Work

First, you must change the value of the RAM pointer, which tells the VIC where to get data to construct the characters you see on the screen. This pointer is memory location 36869. There are a few values that you may Poke in there to reset the VIC to point to your own character set. These are listed in the *VIC Programmer's Reference Guide*. The two most used are 255 and 240. The former will cause the VIC to get its character set from memory locations 7168 to 7679.

But what is a character set? It is nothing more than a group of eight bytes, starting from a certain memory location and extending to some final location. In this case we start at 7168. This and the next seven bytes will define the "at" symbol (@). The next eight bytes define the A symbol, and so forth.

Since I have defined 58 symbols to work with, you'll end up

at 7632 ($7168 + 8 \times 58$).

So the first line of your programs would probably be

1 POKE 36869,255

But you must be careful to protect your character set from variables, which will be stored in the same area of memory as your character set and would therefore destroy the designs you have created. To protect them from variables, you must tell the VIC to lower the top of memory and variables below the character set. The VIC's operating system will then think that you do not wish to use this memory space and will avoid using it.

Memory locations 51, 52, 55 and 56 tell the VIC where the end of memory and the bottom of string storage are located. So, if you Poke in the appropriate values here, you can trick the VIC into thinking it has less memory and, possibly, prevent it from messing up your character set, which is now in this unused area of memory.

What are the numbers to Poke in? To protect memory area 7168 and up, you would divide 7168 by 256. The integer value you get (don't round off!) is the *page* of memory you wish to protect. If you get a remainder, this will be extra memory bytes you wish to protect. In this case, you'll get page 28 with remainder 0.

The remainder will be Poked into 51 and 55 (the low bytes) and the page into 52 and 56 (the high bytes).

1 POKE 51,0: POKE 52,28: POKE 55,0:
POKE,56,28: CLR

Notice the CLR at the end of the line. Its purpose is to reset important page zero pointers. Don't forget it!

Now, the second line can be:

```
5 POKE 36869,255
```

At this point, the screen turns to garbage! This is because you have nothing but random garbage at memory locations 7168 and up. You must put some meaningful data designs here to allow the VIC to design and print your characters properly.

Would you like to be able to use the letters and number designs that you had before? Well, you can simply transfer (or copy) the designs from the character ROM chip (which is where you were getting them before, when memory location 36869 contained 240). The following is a simple loop that will move them for you from the ROM character chip to the RAM area you've chosen (7168 and up).

```
10 FOR D = 0 TO 512
12 POKE 7168 + D, PEEK ( 32768 + D )
14 NEXT D
```

As you run this part of the program, the garbage will quickly turn to meaningful and readable information.

Now for your character set! All you have to do is copy from your data sheet or paper the correct data bytes you've created for your newly designed characters and place them into the new character set RAM.

Suppose you wish to replace the letter A with whatever character you had designed for the

purpose (not a good idea, since we use the letter A so much, but this is only an exercise).

On your data sheet or your paper, you'll have, let's say, 1,255,255,255,255,255,255,255,255. You could have anything, but the first number must be a 1, because this is the character number for the letter A. The next eight numbers can be any number less than 256 and equal to or greater than 0. (This particular set of bytes for the letter A will produce a reversed blank space.)

To transfer this data to its correct position in the character set, use the following loop (and notice the flag - 1).

```
20 RESTORE
23 READ A
25 IF A = - 1 THEN 40
28 FOR B = 0 TO 7
30 READ D
32 POKE 7168 + A*8 + B, D
34 NEXT B
36 GOTO 23
40 REM THIS WILL BE THE REST OF
   YOUR PROGRAM
—
—
999 END
1000
DATA 1,255,255,255,255,255,255,255
1010 DATA - 1
```

Notice that you could easily have used even more user-defined characters. All you must do is place them in the Data statements at the end of the program (but before the - 1 Data statement). Do those the same way as the A character (the first number being the character number and next eight numbers being the design for the character from your data sheet or your paper).

Now, whenever you enter Poke (screen location),¹ you will not get an "A," but you will get your new character. You'll also get your new character if you type PRINT "A".

How to Use Multicolor

To set this space to Multicolor mode, you must Poke the corresponding color memory location with whatever character color you have selected *plus eight*. In this case, you can simply enter Poke (screen location + 30720), (character color + 8). This simple formula will always work and is the simplest way to keep a one-to-one correspondence between your character screen and your color screen. (Actually, this will always work unless you reset the screen or color memory to a different place in memory.)

So, whenever you place a character to the screen in multicolor, first Poke the color memory with the above formula, then Poke the screen memory with this formula: POKE screen location, character number.

Another way to activate Multicolor mode is by printing with a color code greater than 7. Memory location 646 is the location for the current printing color. Normally, it's from 0 to 7, but if you Poke it with a number from 8 to 15, you'll then be printing in Multicolor mode.

The color you will Poke in will be the color character number from the list (0 to 7), plus 8 added to activate the mode.

For example, to begin printing in Multicolor mode using

red as the character color, enter POKE 646, 2 + 8 (the 2 for red and the 8 to activate Multicolor). To cut off multicolor printing, just enter POKE 646 with a number less than 8, or just use a regular color command inside a Print statement.

How About Colors?

To set the four multicolor colors in the VIC, use the following four Pokes.

1. Screen color.

POKE 36879, PEEK (36879) AND
15 OR (SCREEN COLOR * 16)

2. Border color.

POKE 36879, PEEK (36879) AND
248 OR (BORDER COLOR)

Note that the border color must be from 0 to 7 only!

3. Character color. This is individually set for each space on the screen as discussed above. Note that character color is from 0 to 7 only, but you must add 8 to it to activate Multicolor mode in that space on the screen.

4. Auxiliary color.

POKE 36878, PEEK (36878) AND
15 OR (16 * AUXILIARY COLOR)

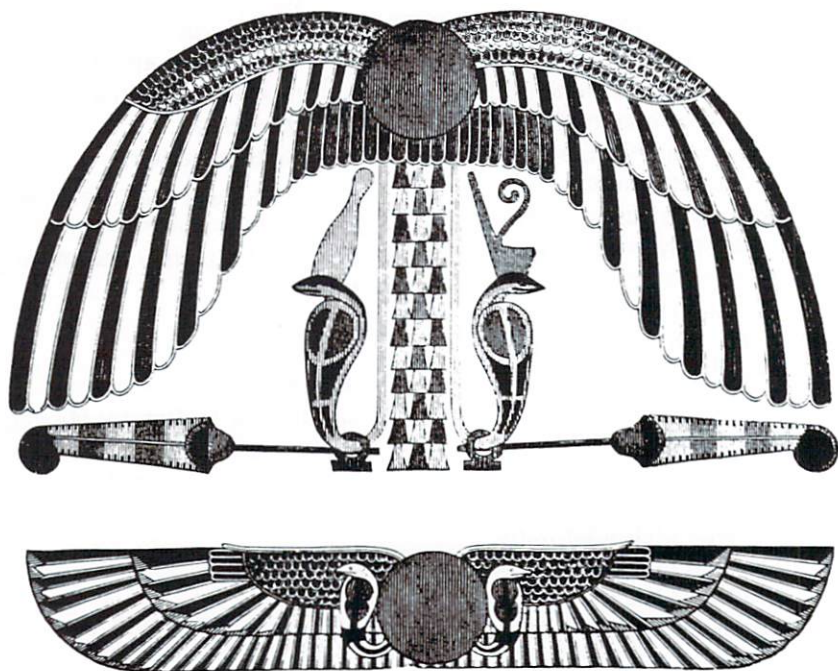
Note that auxiliary colors range from 0 to 15. 

Address all author correspondence to Tommy Michael Tillman, c/o T Squared Software, Box 1133, Sanford, NC 27330.

0—black
1—white
2—red
3—cyan
4—purple
5—green
6—blue
7—yellow

8—orange
9—light orange
10—pink
11—light cyan
12—light purple
13—light green
14—light blue
15—light yellow

Table 1. Color list.





Playing The Ponies

BY GABE GARGIULO

Put away your binoculars and play the horses, through this program and your VIC-20, without losing any money.

This horse racing program for the unexpanded VIC-20 is a conversion of a program written for the PET, found in *Microcomputing* ("Betting on Old POKEY," October 1980). The major change I made is in the graphics for the horse. I used the π symbol, which looks a bit like a horse, or a dog or a chicken, if you use your imagination. My version will work only on the VIC, since it uses Pokes.

You begin the game with \$500, and may bet up to that amount. You pick a horse, numbered from 1 to 5, to win. The program randomly chooses one horse to win. If you pick the winner, you win four times the amount that you bet, which is added to the money that you're holding. If you lose, you lose the amount that you bet. You

play until you lose all the money you're holding. (This is inevitable.)

The game is easy to type in, fun to play, and above all, costs you nothing. The program listing shows a good programming style, which, if adopted, gives you a result that is easy to understand and modify.

Start with a remark showing the program's name and purpose. Then list the variables and explain them. After that, start the main logic of the program, which contains its major decisions. Place the subroutines, which you use with GOSUB statements, after the program's main logic. 

Address author correspondence to Gabe Gargiulo, 26 1/2 New-man St., Manchester, CT 06040.

**Run it
RIGHT**

Unexpanded VIC-20

Line Number

15—	Sets R1 (amount of money held) to 500.
20—	Clears screen. Calls on subroutine 1200 to put row of hyphens across screen. Describes variables ML, LA.
22—	Sets ML (Memory Location) to 7680, the starting address of the upper left of the screen. Sets LA (lines across) to 22. Poke 36879: Sets color and background.
40—	Calls on subroutine 1200 to put a row of hyphens across screen.
80—	Prints the title and calls on subroutine 1200 to put a row of hyphens across screen.
85—	Delays a bit, then starts.
86—	Sets starting position of horses.
90-110—	Give instructions.
140—	Calls on subroutine 1200 to put a row of hyphens across screen.
270—	Gets the number of the horse being bet on.
300—	Calls on subroutine 1200 to put a row of hyphens across screen.
310—	Asks for bet.
320—	If bet is less than or equal to amount held, goes to 400.
330—	(Otherwise) Tells how much is left to bet.
350—	Asks for bet again. (310).
400—	Clears screen and calls on subroutine 2000 to display horses.
410—	Calls on subroutine 4000 to display starting gate.
600—	Gets a random number between 1 and 5.
620—	Calls on subroutine 1000 to add 1 to a counter corresponding to the horse whose number has come up. Calls on subroutine 2000 to put horses on screen. (If a horse's counter has been incremented, its position is advanced.)
630—	Adds 1 to a counter corresponding to the horse whose number has come up.
640—	If the horse that just moved is not near the right side of the screen, goes to 600 to make another horse move.
650—	If a horse has won, falls into here. Delays a bit.
660—	Tells who is the winner.
665—	Prints a row of hyphens across the screen.
670—	If the horse picked is the winning horse, adds the winnings to the amount held, goes to 750.
680—	If the horse picked is not the winner, falls into here. Displays "You lose." Subtracts bet from amount held.
685—	Tells how much money is left.
687—	If no money is left, displays "You're broke." Ends program.
690—	Asks if another game is to be played.
691—	Gets reply.

Table 1. Description of main program.

- 700— If reply is "Y," goes to 20 to start again.
- 710— If reply is not "Y," falls through to here. Restores screen color and background. Ends the program.
- 750— Displays "You win" and how much won.
- 760— Displays amount held.
- 770— Goes to 690 to ask about another game.
- 990— Ends.

Subroutines

- 1000— Adds 1 to X1, X2, X3, X4, or X5, depending on the random number that came up.
- 1200— Puts a row of asterisks across the screen.
- 2000— Advances the horse whose number has come up. Leaves the other horses where they were.
- 3000— Makes the sound of a starting gun and galloping of horses.

Table 2. Descriptions of subroutines.

Variables

- ML— Memory location of horse.
- LA— Lines across screen, 22 for VIC.
- X1— Position of horse 1.
- X2— Position of horse 2.
- X3— Position of horse 3.
- X4— Position of horse 4.
- X5— Position of horse 5.
- R1— Amount of money held.
- I— Index variable.
- R— Random number.
- H— The horse bet on.
- B— Amount of bet.
- B()— Array used to keep track of position of each horse. B(1) is for horse #1, B(2) is for horse #2, etc.
- Z\$— Reply Y/N.
- J— Index variable.
- Y— "Y" coordinate on screen. (How many lines down from top.)
- J3— Index variable.
- Y— Index variable.
- J2— Index variable.
- L— Index variable.
- M— Index variable.

Table 3. Definitions of variables.



Lost In Space

Formerly "Space Rescue."

BY KEN GARDNER

As commander of a mothership in space, can you steer your unmanned drones into a minefield to rescue 18 astronauts who'll soon be gasping for air?

In Space Rescue, a challenging all-graphics game for the unexpanded VIC-20, you are the commander of an interstellar rescue cruiser for the Space Patrol in the year 2090. The cruiser is the mothership to three unmanned drones that are remotely controlled from it.

A space shuttle, carrying 18 astronauts, has collided with an asteroid and drifted into a space minefield. The astronauts ejected from the shuttle, but are floating helplessly in the field.

To fly in and save the astronauts before their air supply runs out is your mission. The mothership is too large to enter the minefield, so you must

send the drones to retrieve the astronauts one by one.

Be warned, don't collide with or shoot at the mothership or that'll be the end. Use the docking bay, located at the bottom center of the cruiser, to drop off astronauts. Also, don't hit the mines or you'll lose a drone and possibly an astronaut.

The entire game is seen from the radar screen on the rescue cruiser. The border lines mark how far the drones can go without flying out of radar range. The top left of the screen shows your score and the top right shows how many drones you have left. Use the joystick to steer the drones through the minefield to pick up the astronauts and return them to the mothership.

When you pick up an astronaut, you'll hear a beep. When you return the astronaut to the mothership, you'll hear a lower-pitched tone. If you lose a drone by flying out of radar range or hitting a mine, you'll hear a high-pitched beep, then one of the drone ships on the top right of the screen will disappear. Remember not to pick up more than one astronaut, because the drones can only transport one at a time.

Each drone is equipped with a photon blaster to clear mines. Press the fire button on the joystick to operate the blaster. You can reload by going back to the mothership.

**Run it
RIGHT**

Unexpanded VIC-20
Joystick

You will have three minutes to rescue the 18 astronauts before they run out of air. At the beginning of play, the computer's internal clock, TI\$, is set to "000000". On line 140, TI\$ is checked to see if more than three minutes have passed. If you'd like more time, you can change the number in quotes on line 140.

Before you start playing, the computer will ask you for a skill level, from one to nine. On the first level, you'll receive one point for each astronaut saved. On the ninth level, you'll receive nine points for each astronaut saved. The higher the skill level, the more credits you'll earn for your work. Your skill level also determines how many mines appear on the screen and how many shots your photon blasters can fire between reloadings.

Type in part one and save it onto tape with part two saved immediately after it. In part one, lines 170-390 and 410-420, which are the instructions, can be omitted to save a little typing.

In the first part of this program, the top of memory is lowered, so there's enough room to put programmable characters. If you use a Programmer's Aid while entering this program, make sure you turn the computer off and on before running the game. The Aid raises the top of memory and messes up some of the special characters. If the characters still look strange, check and double check the data in part one.

Disk users, if you're planning on saving this program on disk, replace line 400 in part one with:

```
400 PRINT CHR$(147)"LOAD"CHR$(34)
  "(Program Name)"CHR$(34)"8":POKE198,
  2:POKE631,19:POKE632,131
```

For his help at the two or three points where I really got stuck while writing this program, I would like to thank my dad, Dave Gardner. 

*Address all author correspondence to Kenneth Gardner,
2342 Barnes Road, Walworth,
NY 14568.*



Part One

10		Lowers top of memory
20		Initializes variables
30-120		Title page
130-140		Poke character information above Basic
150		Reseeds random number generator
160-430		Instructions
440		Title page data
450-610		Character information data

Part Two

10-40		Initialize variables
50-70		Set skill level
90		Places border
100		Places mines and astronauts
130		Reads joystick
140		Checks time
150-190		Set direction for drone or start firing sequence
200-280		Check if drone hit anything
290		Moves drone
300-330		Explosion routines
340-370		Any drones left?
380-410		Play again?
420-460		Drone docks into mothership
470-480		Drone undocks from mothership
490-500		Choose random screen locations for mines and astronauts
510		Fires shot
520-550		Check if shot hit anything
570-580		Move shot
610-620		"You hit the Mothership!" message
630-660		Display score at end of game
670		Timer
680-690		Mothership appears
700-710		Mothership leaves
730		Any drones left?
750		Timer
790		Beep routine
800-810		Update score and ships left
820		Border data

Table. What the lines in the Space Rescue program do.



I Am The President

BY SCOTT CALAMAR

With this satirical program that simulates intelligence, you can throw a party and let your friends talk with a former President of the United States.

When I bought my VIC-20 about a year ago, it was my first hands-on experience with a computer, and although the VIC is exceptionally user-friendly, I didn't think it was friendly at all.

It didn't say hello, didn't ask how I was, didn't even wish me a happy day. The cursor blinked on and off, waiting for me to do something. The machine said it was ready, but I sure wasn't.

Like any good VIC user, I then read my user's manual. I discovered that if I wanted the computer to say hello, I had to make it print hello. I started tagging Print statements onto programs so they would begin on a friendly note. My VIC-20 was

hardly HAL from 2001, but I'd made a start.

By using Get and Input statements, you can simulate intelligence fairly easily. A Get statement will accept a single letter or number and act on that information. An Input statement will accept a string of information—words or a sentence. You can program the computer to recognize that data and respond appropriately.

Create A Conversationalist

Simulated intelligence is used in many programs to accept information, and in video games to find out the number of players, skill level and so on. Although programs that use artificial intelligence are written for some computers with more memory, I've seen few for the VIC-20. The computer's memory constraints make it difficult to include enough alternatives in an intelligence program to be convincing.

I Am the President is a demonstration program that shows how effectively Input and Get statements can simulate intelligence. When you run the program, any expanded VIC-20 will assume the personality of a former President of the United States. You'll be in for a brief meeting with the elder states-

**Run it
RIGHT**

Expanded VIC-20



man, but watch what you say! He's grown very sensitive in recent years.

Just a brief disclaimer: I Am the President is meant as satire—the president is a caricature and not intended to tarnish the memory of any person, living or dead.

Those of you who own VICs with 3K expansion should be careful not to type any additional spaces when entering the program. I Am the President is about 100 bytes short of filling your memory. Users of larger expanders should feel free to modify and add to the program.

I Am the President should provide you with a few moments of entertainment and show your friends what your computer can do.®

Address all author correspondence to Scott Calamar, 917 San Anselmo Ave. #5, San Anselmo, CA 94960.



Nimbots

BY MICHAEL BUCKLEY

You will soon be confronted by a dozen nasty Nimbots, whose single-minded obsession is to preside over your defeat. This cunning game of the mind is both fun and challenging.

In Nimbots, you and the computer—or a human opponent—take turns removing from one to four Nimbots according to certain rules. To move, you key in the letters of the Nimbots that you want taken away and press the return key. Nimbots taken in one turn must be in a straight line, horizontally or diagonally but not vertically, and there must not be any gaps. For example, ADHL would be a legal move, but ADL would not—despite whether or not H is still in place. (The computer will not accept illegal moves.) The player forced to make the last move is the loser.

**Run it
RIGHT**

VIC-20 with 8K expansion

Eight Versions

The above rules describe the standard version. When you've mastered that, you can try the variation in which the object of the game is reversed: You try to take the last Nimbot yourself.

There are also versions in which the no-gaps rule is waived—for example, FHL would be an acceptable move (regardless of the presence of G). The straight-lines rule applies to all versions, however.

All four variations may be played by one or two persons. This gives you a total of eight choices, which you select by pressing the appropriate function key, according to the table below.

Function	No. Players	Gaps	Last Player
F1	one	no	loses
F3	one	no	wins
F5	one	ok	loses
F7	one	ok	wins
F2	two	no	loses
F4	two	no	wins
F6	two	ok	loses
F8	two	ok	wins

In the one-player versions, if you do not wish to make the opening move, simply press the return key, and the computer will go first.

When asked to do so, you must choose a difficulty level from zero to nine. At the higher levels, the computer plays flawlessly—but you can still beat it if you make all the right

moves. At lower levels, the computer often acts randomly.

Save It First

Nimbots is written in Basic, with a machine language subroutine that is Poked into the cassette buffer starting at address 828. To avoid losing an untested program, be sure to save it at least once before you run it.

The total of all the values in the M% array is 49680, and the sum of all the numbers Poked into memory locations 828-1003 is 24627. Before you run Nimbots, enter:

```
25 FORJ = 0 TO 71: T = T + M%(J): NEXT J
PRINT T: END
```

The program should display the number 49680 and stop. If you

get any other number, you have an error somewhere in Data statements 30-100. When you get the correct total, replace line 25 with line 125:

```
125 FORA = 828 TO 1003: T = T + PEEK(A):
NEXT A: PRINT T: END
```

This time, if you don't get 24627, you have a Data error in lines 828-991. When you've got it correct, take out line 125. If you have no other mistakes, you should be able to "RUN It Right." 

Address all author correspondence to Michael R.W. Buckley, 445 East 19th St., North Vancouver, B.C., Canada V7L 2Z6.

Nimbots' Ancestor

Nimbots is one of the many descendants of Nim, a game in which two players take turns removing one or more counters from any one pile. The player unable to make a move is the loser—in other words, the winner takes the last counter or counters.

After playing the game for a while, you begin recognizing certain "safe" positions from which your opponent cannot win. Two identical piles are safe: whatever your opponent does to one pile, you do to the other. Therefore, playing Nim with three piles, containing one, two and three counters is safe because you can always match your opponent's first move and force equal piles.

The VIC-20 is one of the many microcomputers built around the 6502 microprocessor. Included in the 6502 instruction set is the EOR, or "Exclusive OR," instruction. EOR compares two binary numbers, bit by bit, giving a 0 when corresponding bits are the same and a 1 when they differ.

For instance, 89 EOR'ed with 108 would give 53. This example serves to illustrate why the instruction is often called "add without carry."

89 (decimal) = 1011001 (binary)

108 (decimal) = 1101100 (binary)

53 (decimal) = 0110101 (binary)

Obviously, any number EOR'ed with itself is 0. Also, 1 EOR'ed with 2 EOR'ed with 3 is 0.

Before there were computers, mathematicians had another name for this operation: they called it *nim-summing*. The nim-sum of any safe position in Nim is 0!

One version of Nim starts with three piles of three, five and seven counters. You compute the nim-sum to be 1. For example:

3 (decimal) = 011 (binary)

5 (decimal) = 101 (binary)

7 (decimal) = 111 (binary)

1 (decimal) = 001 (binary)

Taking 1 from any pile will reduce the nim-sum to 0. You can't win if you're facing 2, 5, 7 or 3, 4, 7 or 3, 5, 6 (unless your opponent makes a mistake later).

Now comes the switch. Normally this game is played in reverse: You try to make the other player take the last counter. The strategy for this version is left as an exercise for the reader.

How the Program Works

Since there are 12 Nimbots, and each one either is or isn't there, there are 4096 (2 to the 12th power) possible configurations. Each element of the A% array contains a number that tells the computer what move to make if it encounters the corresponding position. If an element contains a 0, then that position is safe (for the opponent) or unanalyzed (in the low-difficulty version), and the computer moves randomly.

Let's set up a sample game board display on which A, F, J, K and L are visible. A%(2119) represents this setup. In binary, 2119 is 100001000111. The alphabet letters A-L run from left to right in this binary number. (Include the leading zeroes so that the resultant 12-digit number will match the 12 alphabetized Nimbots.) If A%(2119) contains 6, which decodes to 000000000110, then the computer would select Nimbots J and K, leaving you with three isolated Nimbots (A, F and L) and certain defeat, assuming the standard no-gaps-last-person-loses version of the game.

How does A% get to contain these values? There are 72 legal moves, including all versions, and they are stored in the M% array. For instance, M%(13) may or may not contain 6, because this array gets shuffled to randomize the play. In binary, 6 is 000000000110, which, as you saw above, stands for Nimbots J and K.

A% is scanned from beginning to end. When an element is found to contain a zero, representing a safe position, then each legal move in M% that could lead to that position is added to the (safe) index, giving the index of an unsafe position. The move in M% is then stored in each location that is computed to be unsafe. Referring to the above example, A%(2113) contains 0—a safe position. There are many moves in M% that could lead to this position, one of which, in M%(13), is 6. Adding 6 to 2113 yields 2119. So 6 gets stored in M%(2119).

Of course, all this is done before you make your first move. It's a procedure that takes over 20 minutes in Basic, but only a couple of seconds with the included machine language subroutine. If you want to compare the two versions of the routine, Listing 2 shows the assembly code alongside the corresponding Basic statements in the comments field.

Aftermath

Here are some questions I had to answer before I could convert those few lines of Basic into machine language. This information would have been invaluable to me a few months ago—I hope it saves somebody else some needless frustration. Reference to Listing 2 will help you understand the answers; some knowledge of the 6502 assembler is assumed.

1. Where are some safe places in zero page for indirect addressing? Nimbots uses locations 163–176. I've used this area without any ill effects so far, but check your memory map to ensure that the system's use of these locations doesn't conflict with yours.

2. How do you get into Basic arrays from machine language? Use the Start of Arrays vector at addresses 47–48. It points to the prologue of the first dimensioned array—the array itself is seven bytes further along. Other arrays occur in order of appearance, each

after a seven-byte offset. In lines 829–853 of the assembly listing, you'll see how I stored the address of the first byte of M%, in 163–164, and of A%, in 165–166.

3. How do you maintain relocatability when you need to jump more than 127 bytes? Use a branch as a stepping stone. Look at lines 916 and 918. They are both BNEs. Obviously, the second one can never be executed under normal circumstances—it's just a dummy instruction. Now look at line 1001. I would like to have put BNE 863 here, but that's beyond the range of relative addressing. So, instead, I put BNE 918, and then at 918, I inserted the BNE 863 right below another BNE.

Finally, I'd be interested in hearing from readers who find a simple strategy for any version of Nimbots. ☐



Spelling Friend

BY WILLIAM W. BRAUN

Does your youngster need help in learning his/her weekly list of spelling words? Well, meet Chippy, who's the best spelling buddy your child could have.

In Spelling Friend, your child can practice spelling with a simulated computer friend, Chippy. My daughters, ages nine and six, enjoy using the program to study their weekly spelling assignments. Even the six-year-old is now able to enter her weekly list of words.

Chippy, who appears as a large smiling face with curly hair, first shows you the list of spelling words contained in his memory. He then asks you if the words are all right for the current spelling session. If you answer no, then Chippy tells you that you must type in 20 new words and prompts you when to do so. After you've entered the 20 words, Chippy displays the new word list, asking

if those are all right. You may repeat this process until satisfied.

When you indicate that the words are all right, Chippy presents the program's instructions. He indicates that a word will be displayed for a few seconds. After it vanishes, Chippy will ask you to type it in correctly. After the instructions are displayed, you are given the option of seeing them again or continuing.

When you choose to continue, Chippy shows the first word in his list. The word, which is enclosed in a multi-color border, appears one letter at a time. Each letter is accompanied by a short tone, which increases in pitch with each letter. After the word disappears, you must try to type it in from memory. If you succeed, Chippy appears with a big smile, gives a short message of encouragement and winks at you. If you spell the word incorrectly, Chippy frowns and instructs you to try again. If you spell the word incorrectly twice, Chippy shows you the correct spelling. Periodically during the program, at least some of the initially misspelled words will be shown again, giving you more practice with them.

After all 20 words have been used, Chippy shows you your score and gives a message

**Run it
RIGHT**

VIC-20 with 3K expansion

about your performance. At this point, you may choose to start over again, see a list of the words you misspelled or end the program. If you choose to stop, Chippy informs you that you may resave the program if you want to have the same words for the next practice session.

About the Program

Through Chippy, I tried to create a feeling of personal communication between the child and the computer; the computer is no longer only a machine that displays words and responds negatively or positively to a child's input—the computer has a personality.

If you have a speech synthesizer, you can replace or supplement the messages on the screen with verbal statements from Chippy.

Unfortunately, I could not code this program to run on the unexpanded VIC; I would have had to sacrifice most of the features that make it interesting. I coded it so that you can use it with any amount of expansion. Some programs will run only if a particular amount of RAM is present. This is because the VIC operating system changes the screen and color memory locations when you add more than 3K expansion. If your program does not take this into account and provide for variable screen and color memory locations, you must run the program on a VIC with a specific RAM configuration. This can get very frustrating if

you have a variety of programs, and it can be rough on the expansion port connectors as you switch around the RAM expansion cartridges.

In Spelling Friend, line 9100 takes care of this problem. The program Peaks location 44, which will hold the number 18 if the VIC has more than 3K of memory expansion. It then chooses the proper screen and color memory constants, which are based on the result of the Peek. If your programs will be Poking things around the screen, you'll save yourself and others a lot of trouble by including this option in your programs.

You can often save yourself a lot of coding if you create subroutines to handle repetitive tasks. This program uses many subroutines. For example, there are routines to create Chippy's smiling or frowning face, to make sound effects, to produce delays in the program action, to respond to correct and incorrect spelling inputs and to create the multicolor border around the spelling words.

Kids love to play with the keyboard, just to see what will happen. This can be a problem if they decide to try out the run/stop key in the middle of a program. Line 6 anticipates this problem, and by Poking 114 to location 808, it turns off the run/stop key. The restore key doesn't become disabled, since the child would have to press the run/stop and restore keys at the same time, which would be more unlikely to happen.

Programming Techniques

While writing Spelling Friend, I had to find a way to prevent the child from typing in the word while it was still being displayed on the screen. When the Input statement was executed after the word vanished, the program would use the word in the keyboard buffer. To prevent this, it finally occurred to me to use POKE 198,0 to clear the keyboard buffer, immediately after the word disappears and before the actual Input statement is executed. This occurs in line 61. The child may now type in the word while it is being displayed, but it will not be picked up by the Input statement, and the child will have to reenter it after the word disappears.

New words are placed in Chippy's spelling list by utilizing the "dynamic keyboard" technique. Lines 463-468 contain the routine that creates new Data statements with the new words. As the new words are entered, they are placed into an array, NW\$.

Five lines, beginning with number 9000, are then printed on the screen. These lines are Data statements, which contain the new words. A sixth line, without a line number, is printed on the screen. This last line defines a variable and has a command to go to line 9100. The cursor is moved to the Home position. The CHR\$ code for RETURN, 13, is then Poked into the keyboard buffer six times. When the End statement is reached, in line 468, the six

returns in the keyboard buffer are executed, putting the new Data statements into the program (while erasing the old ones) and executing the GOTO 9100 command.

The variable VB is used in line 9117, to decide whether or not the program should continue at line 6 or line 20. The first time the program is executed, it goes through lines 9100-9120 to initialize variables, and then returns to line 6. The only other time line 9100 is executed is if new words are being entered into the program. At this point, it is necessary to start at 9100, since the program actually ended (albeit only for a split second) in line 468; however, this time you jump to line 20, since you needn't go to line 6 to see the program title screen again.

The only other way I could find to change the spelling-word list was to actually exit the program and type in new Data statements, and then restart the program from the beginning. The dynamic-keyboard technique, which you can probably find many uses for in your own programs, is much cleaner and easier to use, especially for children. It can also be used to place commands in the keyboard, to erase the current program and load and run another program. If you put two programs, one after the other, on tape, you can use this technique to load and run the second one as the first is ending. This would be very useful if your program exceeded 3.5K, as you can split it

up into two sections. You can even load a third program when the second is completed, and so on. With this trick, you can make the VIC run some very long programs, providing they are of a type that can be split up into parts.

Looking at lines 9110-9112, you will notice that several string variables have been defined as being equivalent to CHR\$ commands. I use these string variables right after a Print statement, to execute the CHR\$ commands, which perform the same function as familiar keyboard programming commands. For example, CD\$ is equivalent to moving the cursor down one line; BLK\$ changes the print color to black; CH\$ clears the screen and moves the cursor home; and LC\$ changes the characters to upper/lowercase. This technique produces listings that are much easier to understand. Instead of getting confusing graphics symbols, you get easy-to-read string variables.

The string variables, with their well-chosen names, make it much easier for you to remember their functions. Defining string variables in this manner also clarifies which symbols in a listing are commands and which are actually graphics characters. The only drawback I have found to using this method is that each defined string variable eats up a good chunk of memory. If you are confined to the unexpanded VIC, it could use up too much memory.

Lines 200-205 contain subroutines to produce delays of varying length. Rather than writing out a For...Next statement each time I want a delay, I simply call up the appropriate subroutine. I put longer delays in consecutively higher line numbers, to make it a bit easier to remember which line the GO-SUB accesses. If your program will call for using the same length delay repeatedly, this method can save you time and bytes.

Making programs as user friendly as possible is an important aspect of programming. This includes trying to anticipate problems with Input statements. For example, lines 46-49 control the program's response to the child's input as to whether or not the instructions should be repeated or the spelling words commence.

I chose to use a Get statement rather than an Input statement since only one key needs to be pressed. Line 46 freezes the action until a key is actually pressed. Line 47 checks to see if the S key was pressed and it takes appropriate action. Line 48 watches for the I key to be pressed; if it has, it repeats the instructions. If any key other than S or I is pressed, the program falls through to line 49, which prints an error message to the screen that informs the child that he or she can enter only I or S, and then branches back to line 46.

The same technique of editing the input is used in lines 418-430, but this time

with an Input statement. This type of editing simply ensures that the program is not stopped with an inappropriate input, and makes it clear just what input is actually needed. $\square$

Address all author correspondence to William W. Braun, 3164 Wellington Way, Arnold, MO 63010.

Program documentation

Line number(s)	Comments
6-17	Title screen graphics and sound effects (subroutine at 9300-9380 draws the border with letters of the alphabet).
20	Dimensions arrays to hold spelling words and incorrectly spelled words. A\$ is for spelling words and W\$ is for incorrectly spelled words. Reads Data statements to fill A\$ array.
25-30	Initialize variables, set border/screen colors.
39-49	Instructions to student.
50	Variable A counts number of spelling words displayed. If A equals 20, program branches to give score.
57	Variable P is used to display incorrectly spelled words a second time.
75	Detects incorrectly spelled word.
77	Detects correctly spelled word.
550-573	Print Chippy's faces. Value of FA decides if frown or smile.
600-620	Routine to show correct spelling after two wrong answers.
700-729	Display score and decide upon message about student progress based upon score.
1990-2005	Routine to print spelling words to screen, one letter at a time with ascending tones and centered in the multicolored box.
3016	Prints list of incorrectly spelled words.
6000-6001	Sound effects and border colors with correctly spelled words.
6200-6210	Blink Chippy's eye.
6500	Buzzing sound with incorrect answer.
8000-8003	Create multicolor border around spelling words.

Definitions of variables

SM—Screen memory location
 CM—Color memory location
 R—Number of words spelled correctly
 A—Subscript of A\$(A), number of words displayed
 W—Number of words spelled wrong
 C—Variable to detect two incorrect spellings in a row
 B\$—Variable to hold typed-in spelling words
 W\$—Incorrectly spelled words
 FA—Value decides if Chippy has smile or frown
 T—Variable in delay routines



Please send me ReRUN Volume I!

___ **Cassette version(s) at \$11.47* each.**

___ **Disk version(s) at \$21.47 each.**

* Prices include \$1.50 postage and handling.
Foreign Air Mail please add an additional 45¢ per item.
U.S. funds drawn on U.S. banks ONLY.

☐ **Check/MO** ☐ **MC** ☐ **VISA** ☐ **AE**

Card # _____ **Exp. Date** _____

Signature _____

Name _____

Address _____

City _____ **State** _____ **Zip** _____

ReRUN • 80 Pine Street • Peterborough, NH • 03458

23 Great New Software Programs For Your Commodore!



If any manufacturing defect becomes apparent within 30 days of purchase, the defective cassette/disk will be replaced free of charge subject to its return by the consumer by prepaid mail. Send a letter specifying the defect to:

ReRUN • 80 Pine Street • Peterborough, NH 03458

Replacements will not be made if the cassette/disk has been altered, repaired, or is misused through negligence, shows signs of excessive wear or is damaged by equipment.

ReRUN is simply the listing from RUN Magazine. It will not run under all system configurations. Use the Key Box accompanying each article as your guide.

The entire contents are copyrighted 1984 by CW Communications/Peterborough. Unauthorized duplication is a violation of applicable laws.



© Copyright 1984 CW Communications/Peterborough

CW COMMUNICATIONS/PETERBOROUGH